

Table of Contents

Introduction	1.1
Getting Started	1.2
Initial Configuration	1.2.1
Toolchain Installation	1.2.2
Mac OS	1.2.2.1
Linux	1.2.2.2
Advanced Linux	1.2.2.2.1
Windows	1.2.2.3
Building the Code	1.2.3
Contributing & Dev Call	1.2.4
GIT Examples	1.2.4.1
Concepts	1.3
Flight Modes V Operation	1.3.1
Architectural Overview	1.3.2
Flight Stack	1.3.3
Middleware	1.3.4
Mixing and Actuators	1.3.5
PWM limit state machine	1.3.6
Tutorials	1.4
Ground Control Station	1.4.1
Writing an Application	1.4.2
Video streaming in QGC	1.4.3
Optical Flow and LIDAR	1.4.4
Integration Testing	1.4.5
Optical Flow Outdoors	1.4.6
ecl EKF	1.4.7
Preflight Checks	1.4.8
Simulation	1.5
Basic Simulation	1.5.1
Gazebo Simulation	1.5.2

HITL Simulation	1.5.3
Interfacing to ROS	1.5.4
Autopilot Hardware	1.6
Crazyflie 2.0	1.6.1
Intel Aero	1.6.2
Pixfalcon	1.6.3
Pixhawk	1.6.4
Pixracer	1.6.5
Raspberry Pi	1.6.6
Snapdragon Flight	1.6.7
 Optical Flow	1.6.7.1
 Snapdragon Advanced	1.6.7.2
 Accessing I/O Data	1.6.7.2.1
 Camera and optical flow	1.6.7.2.2
Middleware and Architecture	1.7
 uORB Messaging	1.7.1
 MAVLink Messaging	1.7.2
 Daemons	1.7.3
 Driver Framework	1.7.4
Airframes	1.8
 Unified Codebase	1.8.1
 Adding a new Airframe	1.8.2
 Multicopters	1.8.3
 Motor Map	1.8.3.1
 QAV 250 Racer	1.8.3.2
 Matrice 100	1.8.3.3
 QAV-R	1.8.3.4
 Planes	1.8.4
 Wing Wing Z-84	1.8.4.1
 VTOL	1.8.5
 VTOL Testing	1.8.5.1
 TBS Caipiroshka	1.8.5.2
 Boats, Submarines, Blimps, Rovers	1.8.6
Companion Computers	1.9

Pixhawk family companion	1.9.1
Robotics using DroneKit	1.10
DroneKit usage	1.10.1
Robotics using ROS	1.11
Offboard Control from Linux	1.11.1
ROS Installation on RPi	1.11.2
MAVROS (MAVLink on ROS)	1.11.3
MAVROS offboard example	1.11.4
External Position Estimation	1.11.5
Gazebo Octomap	1.11.6
Sensor and Actuator Buses	1.12
I2C Bus	1.12.1
SF1XX lidar	1.12.1.1
UAVCAN Bus	1.12.2
UAVCAN Bootloader	1.12.2.1
UAVCAN Firmware Upgrades	1.12.2.2
UAVCAN Configuration	1.12.2.3
PWM V GPIO	1.12.3
UART	1.12.4
uLanding Radar	1.12.4.1
Debugging and Advanced Topics	1.13
FAQ	1.13.1
System Console	1.13.2
System Boot	1.13.3
Parameters & Configs	1.13.4
Autopilot Debugging	1.13.5
Simulation Debugging	1.13.6
Sending Debug Values	1.13.7
Indoor V Fake GPS	1.13.8
Camera Trigger	1.13.9
Logging	1.13.10
Flight Log Analysis	1.13.11
EKF Log Replay	1.13.12

Installing driver for Intel RealSense R200	1.13.13
Parrot Bebop	1.13.14
Gimbal (Mount) Control Setup	1.13.15
Switching State Estimators	1.13.16
Out-of-tree Modules	1.13.17
ULog File Format	1.13.18
Licenses	1.13.19
Software Update	1.14
STM32 Bootloader	1.14.1
Testing and CI	1.15
Docker Containers	1.15.1
Continuous Integration	1.15.2
Jenkins Continuous Integration	1.15.2.1

PX4 Development Guide

Info Developers only! This guide is under active development and not intended for consumers.

This guide describes how to work inside the PX4 system architecture. It enables developers to:

- Get an [overview of the system](#)
- Access and modify the [PX4 Flight Stack](#) and [PX4 Middleware](#)
- Deploy PX4 on [Snapdragon Flight](#), [Pixhawk](#) and [Pixfalcon](#)

License

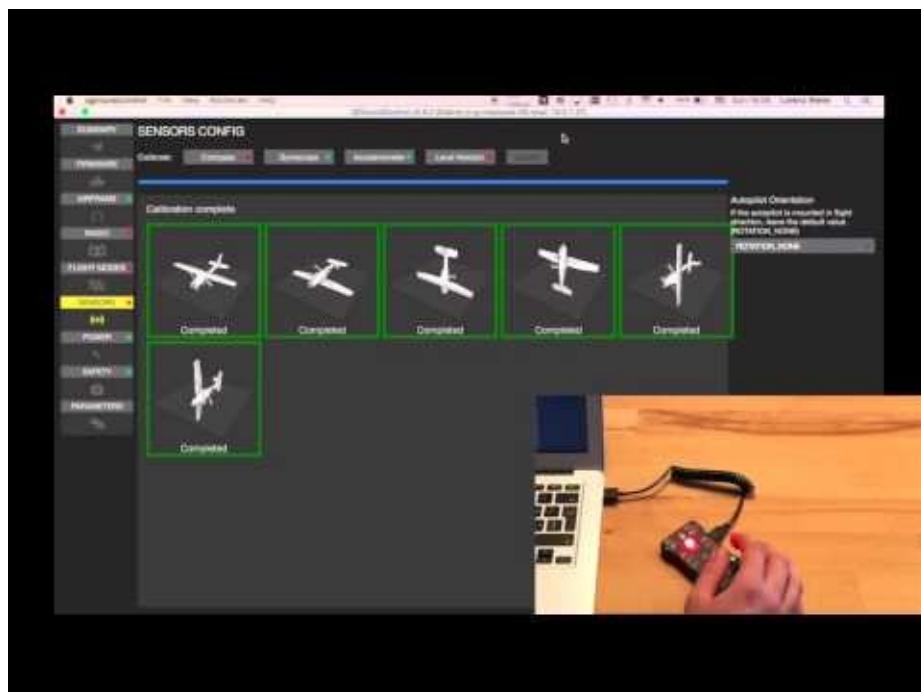
The PX4 Development Guide is available under [CC BY 4.0](#) license. See our [Github Repository](#) for more details.

Initial Configuration

Before starting to develop on PX4, the system should be configured initially with a default configuration to ensure the hardware is set up properly and is tested. The video below explains the setup process with [Pixhawk hardware](#) and [QGroundControl](#). A list of supported reference airframes is [here](#).

Info Download the [DAILY BUILD](#) of QGroundControl and follow the video instructions below to set up your vehicle. See the [QGroundControl Tutorial](#) for details on mission planning, flying and parameter setting.

A list of setup options is below the video.



[Video link](#)

Radio Control Options

The PX4 flight stack does not mandate a radio control system. It also does not mandate the use of individual switches for selecting flight modes.

Flying without Radio Control

All radio control setup checks can be disabled by setting the parameter `COM_RC_IN_MODE` to `1`. This will not allow manual flight, but e.g. flying in

Single Channel Mode Switch

Instead of using multiple switches, in this mode the system accepts a single channel as mode switch. This is explained in the [legacy wiki](#).

Installing Files and Code

The PX4 code can be developed on [Mac OS](#), [Linux](#) or [Windows](#). Mac OS and Linux are recommended since image processing and advanced navigation cannot be easily developed on Windows. If unsure, new developers should default to Linux and the current [Ubuntu LTS edition](#).

Development Environment

The installation of the development environment is covered below:

- [Mac OS](#)
- [Linux](#)
- [Windows](#)

If you're familiar with Docker you can also use one of the prepared containers: [Docker Containers](#)

Once finished, continue to the [build instructions](#).

Installing Files and Code

The first step is to install Xcode from the Mac app store. Once its installed, open a new terminal and install the command line tools:

```
xcode-select --install
```

Homebrew Installation

Usage of the [Homebrew package manager](#) for Mac OS X is recommended. The installation of Homebrew is quick and easy: [installation instructions](#).

After installing Homebrew, copy these commands to your shell:

```
brew tap PX4/homebrew-px4
brew tap osrf/simulation
brew update
brew install git bash-completion genromfs kconfig-frontends gcc-arm-none-eabi
brew install astyle cmake ninja
# simulation tools
brew install ant graphviz sdformat3 eigen protobuf
brew install homebrew/science/opencv
```

Then install the required python packages:

```
sudo easy_install pip
sudo pip install pyserial empy
```

Java for jMAVSim

If you're intending to use jMAVSim, you need to install [Java JDK 8](#).

Snapdragon Flight

Developers working on Snapdragon Flight should use an Ubuntu VM for the time being and follow the Linux instructions. Qualcomm provides reliable tooling for Ubuntu exclusively. The PX4 dev team had the most consistent experience with VMWare, in particular when it comes to USB stability.

Simulation

OS X comes with CLANG pre-installed. No further installation steps are required.

Editor / IDE

And finally download and install the Qt Creator app: [Download](#)

Now continue to run the [first build!](#)

Development Environment on Linux

We have standardized on Debian / Ubuntu LTS as the supported Linux distribution, but [boutique distribution instructions](#) are available for Cent OS and Arch Linux.

Permission Setup

Warning Never ever fix permission problems by using 'sudo'. It will create more permission problems in the process and require a system reinstallation to fix them.

The user needs to be part of the group "dialout":

```
sudo usermod -a -G dialout $USER
```

And then you have to logout and login again, as this is only changed after a new login.

Installation

Update the package list and install the following dependencies for all PX4 build targets. PX4 supports four main families:

- NuttX based hardware: [Pixhawk](#), [Pixfalcon](#), [Pixracer](#), [Crazyflie](#), [Intel Aero](#)
- Snapdragon Flight hardware: [Snapdragon](#)
- Linux-based hardware: [Raspberry Pi 2/3](#), [Parrot Bebop](#)
- Host simulation: [jMAVSIM SITL](#) and [Gazebo SITL](#)

Info Install the [Ninja Build System](#) for faster build times than with Make. It will be automatically selected if installed.

```
sudo add-apt-repository ppa:george-edison55/cmake-3.x -y
sudo apt-get update
sudo apt-get install python-argparse git-core wget zip \
    python-empy qtcreator cmake build-essential genromfs -y
# simulation tools
sudo apt-get install ant protobuf-compiler libeigen3-dev libopencv-dev openjdk-8-jdk \
    openjdk-8-jre clang-3.5 lldb-3.5 -y
```

NuttX based hardware

Ubuntu comes with a serial modem manager which interferes heavily with any robotics related use of a serial port (or USB serial). It can be uninstalled without side effects:

```
sudo apt-get remove modemmanager
```

Update the package list and install the following dependencies. Packages with specified versions should be installed with this particular package version.

```
sudo apt-get install python-serial openocd \
    flex bison libncurses5-dev autoconf texinfo build-essential \
    libftdi-dev libtool zlib1g-dev \
    python-empy -y
```

Make sure to remove leftovers before adding the arm-none-eabi toolchain.

```
sudo apt-get remove gcc-arm-none-eabi gdb-arm-none-eabi binutils-arm-none-eabi gcc-arm
-embedded
sudo add-apt-repository --remove ppa:team-gcc-arm-embedded/ppa
```

Then follow the [toolchain installation instructions](#) to install the arm-none-eabi toolchain version 4.9 or 5.4 manually.

Snapdragon Flight

Toolchain installation

```
sudo apt-get install android-tools-adb android-tools-fastboot fakechroot fakeroot unzi
p xz-utils wget python python-empy -y
```

```
git clone https://github.com/ATLFlight/cross_toolchain.git
```

Get the Hexagon SDK 3.0 from QDN:

<https://developer.qualcomm.com/download/hexagon/hexagon-sdk-v3-linux.bin>

This will require a QDN login. You will have to register if you do not already have an account.

Now move the following files in the download folder of the cross toolchain as follows:

```
mv ~/Downloads/hexagon-sdk-v3-linux.bin cross_toolchain/downloads
```

Install the toolchain and SDK like this:

```
cd cross_toolchain
./installv3.sh
cd ..
```

Follow the instructions to set up the development environment. If you accept all the install defaults you can at any time re-run the following to get the env setup. It will only install missing components.

After this the tools and SDK will have been installed to "\$HOME/Qualcomm/...". Append the following to your ~/.bashrc:

```
export HEXAGON_SDK_ROOT="${HOME}/Qualcomm/Hexagon_SDK/3.0"
export HEXAGON_TOOLS_ROOT="${HOME}/Qualcomm/HEXAGON_Tools/7.2.12/Tools"
export PATH="${HEXAGON_SDK_ROOT}/gcc-linaro-4.9-2014.11-x86_64_arm-linux-gnueabihf_lin
ux/bin:$PATH"
```

Load the new configuration:

```
source ~/.bashrc
```

Sysroot Installation

A sysroot is required to provide the libraries and header files needed to cross compile applications for the Snapdragon Flight applications processor.

The qrlSDK sysroot provies the required header files and libraries for the camera, GPU, etc.

Download the file [Flight_3.1.1_qrlSDK.zip](#) and save it in `cross_toolchain/download/`.

```
cd cross_toolchain
unset HEXAGON_ARM_SYSROOT
./qrlinux_sysroot.sh
```

Append the following to your ~/.bashrc:

```
export HEXAGON_ARM_SYSROOT=${HOME}/Qualcomm/qrlinux_v3.1.1_sysroot
```

Load the new configuration:

```
source ~/.bashrc
```

For more sysroot options see [Sysroot Installation](#)

Update ADSP firmware

Before building, flashing and running code, you'll need to update the [ADSP firmware](#).

References

There is an external set of documentation for Snapdragon Flight toolchain and SW setup and verification: [ATLFlightDocs](#)

Messages from the DSP can be viewed using mini-dm.

```
 ${HEXAGON_SDK_ROOT}/tools/debug/mini-dm/Linux_Debug/mini-dm
```

Note: Alternatively, especially on Mac, you can also use [nano-dm](#).

Raspberry Pi hardware

Developers working on Raspberry Pi hardware should download the RPi Linux toolchain from below. The installation script will automatically install the cross-compiler toolchain. If you are looking for the *native* Raspberry Pi toolchain to compile directly on the Pi, see [here](#)

```
git clone https://github.com/pixhawk/rpi_toolchain.git
cd rpi_toolchain
./install_cross.sh
```

You will be required to enter your password for toolchain installation to complete successfully.

You can pass a different path to the installer script if you wouldn't like to install the toolchain to the default location of `/opt/rpi_toolchain`. Run `./install_cross.sh <PATH>`. The installer will automatically configure required environment variables as well.

Finally, run the following command to update the environmental variables:

```
source ~/.profile
```

Parrot Bebop

Developers working with the Parrot Bebop should install the RPi Linux Toolchain. Follow the description under [Raspberry Pi hardware](#).

Next, install ADB.

```
sh sudo apt-get install android-tools-adb -y`
```

Finishing Up

Now continue to run the [first build!](#)

Linux Installation Instructions for Arch and CentOS

USB Device Configuration

Linux users need to explicitly allow access to the USB bus for JTAG programming adapters.

For Archlinux: replace the group plugdev with uucp in the following commands

Run a simple ls in sudo mode to ensure the commands below succeed:

```
sudo ls
```

Then with sudo rights temporarily granted, run this command:

```
cat > $HOME/rule.tmp <<_EOF
# All 3D Robotics (includes PX4) devices
SUBSYSTEM=="usb", ATTR{idVendor}=="26AC", GROUP="plugdev"
# FTDI (and Black Magic Probe) Devices
SUBSYSTEM=="usb", ATTR{idVendor}=="0483", GROUP="plugdev"
# Olimex Devices
SUBSYSTEM=="usb", ATTR{idVendor}=="15ba", GROUP="plugdev"
_EOF
sudo mv $HOME/rule.tmp /etc/udev/rules.d/10-px4.rules
sudo /etc/init.d/udev restart
```

User needs to be added to the group plugdev:

```
sudo usermod -a -G plugdev $USER
```

Installation Instructions for Uncommon Linux Systems

CentOs

The build requires Python 2.7.5. Therefore as of this writing Centos 7 should be used. (For earlier Centos releases a side-by-side install of python v2.7.5 may be done. But it is not recommended because it can break yum.)

The EPEL repositories are required for openocd libftdi-devel libftdi-python

```
wget https://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-5.noarch.rpm  
sudo yum install epel-release-7-5.noarch.rpm  
yum update  
yum groupinstall "Development Tools"  
yum install python-setuptools  
easy_install pyserial  
easy_install pexpect  
yum install openocd libftdi-devel libftdi-python python-argparse flex bison-devel ncurses-devel ncurses-libs autoconf texinfo libtool zlib-devel cmake
```

Note: You may want to also install python-pip and screen

Additional 32 bit libraries

Once the arm toolchain is installed test it with:

```
arm-none-eabi-gcc --version
```

If you receive the following message

```
bash: gcc-arm-none-eabi-4_7-2014q2/bin/arm-none-eabi-gcc: /lib/ld-linux.so.2: bad ELF interpreter: No such file or directory
```

Then you will also need to install other 32-bit libraries glibc.i686 ncurses-libs.i686

```
sudo yum install glibc.i686 ncurses-libs.i686
```

Pulling in ncurses-libs.i686 will pull in most of the other required 32 bit libraries. Centos 7 will install most all the PX4 related devices without the need for any added udev rules. The devices will be accessible to the predefined group ' dialout'. Therefore any references to adding udev rules can be ignored. The only requirement is that your user account is a member of the group 'dial out'

Arch Linux

```
sudo pacman -S base-devel lib32-glibc git-core python-pyserial zip python-empy
```

Install [yaourt](#), the package manager for the [Arch User Repository \(AUR\)](#).

Then use it to download, compile and install the following:

```
yaourt -S genromfs
```

Permissions

The user needs to be added to the group "uucp":

```
sudo usermod -a -G uucp $USER
```

After that, logging out and logging back in is needed.

Log out and log in for changes to take effect! Also remove the device and plug it back in!**

Toolchain Installation

Execute the script below to either install GCC 4.9 or 5.4:

GCC 4.9:

```
pushd .
cd ~
wget https://launchpad.net/gcc-arm-embedded/4.9/4.9-2015-q3-update/+download/gcc-arm-none-eabi-4_9-2015q3-20150921-linux.tar.bz2
tar -jxf gcc-arm-none-eabi-4_9-2015q3-20150921-linux.tar.bz2
exportline="export PATH=$HOME/gcc-arm-none-eabi-4_9-2015q3/bin:\$PATH"
if grep -Fxq "$exportline" ~/.profile; then echo nothing to do ; else echo $exportline
>> ~/.profile; fi
. ~/.profile
popd
```

GCC 5.4:

```
pushd .
cd ~
wget https://launchpad.net/gcc-arm-embedded/5.0/5-2016-q2-update/+download/gcc-arm-none-eabi-5_4-2016q2-20160622-linux.tar.bz2
tar -jxf gcc-arm-none-eabi-5_4-2016q2-20160622-linux.tar.bz2
exportline="export PATH=$HOME/gcc-arm-none-eabi-5_4-2016q2/bin:\$PATH"
if grep -Fxq "$exportline" ~/.profile; then echo nothing to do ; else echo $exportline
>> ~/.profile; fi
. ~/.profile
popd
```

If using Debian Linux, run this command:

```
sudo dpkg --add-architecture i386
sudo apt-get update
```

and Install the 32 bit support libraries (this might fail and can be skipped if running a 32 bit OS):

```
sudo apt-get install libc6:i386 libgcc1:i386 libstdc++5:i386 libstdc++6:i386
sudo apt-get install gcc-4.6-base:i386
```

Ninja Build System

Ninja is faster than Make and the PX4 CMake generators support it. Unfortunately Ubuntu carries only a very outdated version at this point. To install a recent version of [Ninja](#), download the binary and add it to your path:

```
mkdir -p $HOME/ninja
cd $HOME/ninja
wget https://github.com/martine/ninja/releases/download/v1.6.0/ninja-linux.zip
unzip ninja-linux.zip
rm ninja-linux.zip
exportline="export PATH=$HOME/ninja:\$PATH"
if grep -Fxq "$exportline" ~/.profile; then echo nothing to do ; else echo $exportline
>> ~/.profile; fi
. ~/.profile
```

Troubleshooting

Version Test

Enter:

```
arm-none-eabi-gcc --version
```

The output should be something similar to:

```
arm-none-eabi-gcc (GNU Tools for ARM Embedded Processors) 4.7.4 20140401 (release) [ARM/embedded-4_7-branch revision 209195]
Copyright (C) 2012 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

If you get:

```
arm-none-eabi-gcc --version  
arm-none-eabi-gcc: No such file or directory
```

make sure you have the 32bit libs installed properly as described in the installation steps.

Windows Installation Instructions

Warning Although a Windows toolchain is available, its not officially supported and we discourage its use. It is unbearably slow during Firmware compilation and does not support new boards like Snapdragon Flight. It also cannot run the standard robotics software packages many developers use to prototype computer vision and navigation. Before starting to develop on Windows, consider installing a dual-boot environment with [Ubuntu](#).

Development Environment Installation

Download and install these on your system:

- [Qt Creator IDE](#)
- [PX4 Toolchain Installer v14 for Windows Download](#) (32/64 bit systems, complete build system, drivers)
- [PX4 USB Drivers](#) (32/64 bit systems)

Now continue to run the [first build!](#)

NEW! Bash on Windows

There is a new option for Windows users which is to run Bash shell natively then follow the Linux build instructions. See [BashOnWindows](#). We have verified that the PX4 build succeeds in this environment. It cannot yet flash the firmware, but you can use the Mission Planner or QGroundControl to flash custom firwmare on Windows.

Building PX4 Software

PX4 can be built on the console or in a graphical development environment / IDE.

Compiling on the Console

Before moving on to a graphical editor or IDE, it is important to validate the system setup. Do so by bringing up the terminal. On OS X, hit ⌘-space and search for 'terminal'. On Ubuntu, click the launch bar and search for 'terminal'. On Windows, find the PX4 folder in the start menu and click on 'PX4 Console'.



The terminal starts in the home directory. We default to '~/.src/Firmware' and clone the upstream repository. Experienced developers might clone [their fork](#) instead.

```
mkdir -p ~/.src
cd ~/.src
git clone https://github.com/PX4/Firmware.git
cd Firmware
git submodule update --init --recursive
cd ..
```

Now its time to build the binaries by compiling the source code. But before going straight to the hardware, a [simulation run](#) is recommended as the next step. Users preferring to work in a graphical development environment should continue with the next section.

NuttX / Pixhawk based boards

```
cd Firmware  
make px4fmu-v2_default
```

Note the syntax: 'make' is the build tool, 'px4fmu-v2' is the hardware / autopilot version and 'default' is the default configuration. All PX4 build targets follow this logic. A successful run will end with this output:

```
[100%] Linking CXX executable firmware_nuttx  
[100%] Built target firmware_nuttx  
Scanning dependencies of target build_firmware_px4fmu-v2  
[100%] Generating nuttx-px4fmu-v2-default.px4  
[100%] Built target build_firmware_px4fmu-v2
```

By appending 'upload' to these commands the compiled binary will be uploaded via USB to the autopilot hardware:

```
make px4fmu-v2_default upload
```

A successful run will end with this output:

```
Erase : [=====] 100.0%  
Program: [=====] 100.0%  
Verify : [=====] 100.0%  
Rebooting.  
  
[100%] Built target upload
```

Raspberry Pi 2/3 boards

The command below builds the target for Raspbian.

Cross-compiler build

```
cd Firmware  
make posix_rpi_cross # for cross-compiler build
```

The "px4" executable file is in the directory build_posix_rpi_cross/src/firmware/posix. Make sure you can connect to your RPi over ssh, see [instructions how to access your RPi](#).

Then set the IP (or hostname) of your RPi using:

```
export AUTOPILOT_HOST=192.168.X.X
```

And upload it with:

```
cd Firmware  
make posix_rpi_cross upload # for cross-compiler build
```

Then, connect over ssh and run it with (as root):

```
sudo ./px4 px4.config
```

Native build

If you're building *directly* on the Pi, you will want the native build target (posix_rpi_native).

```
cd Firmware  
make posix_rpi_native # for native build
```

The "px4" executable file is in the directory build_posix_rpi_native/src/firmware posix. Run it directly with:

```
sudo ./build_posix_rpi_native/src/firmware/posix/px4 ./posix-configs/rpi/px4.config
```

A successful build followed by executing px4 will give you something like this:

```
____ \ \ / / / |  
| _/ / \ v / / / |  
| _/ / \ \ / /_ |  
| | / / ^ \ \ \ |  
\| \ \ \ \ \ / / |
```

```
px4 starting.
```

```
pxh>
```

Autostart

To autostart px4, add the following to the file `/etc/rc.local` (adjust it accordingly if you use native build), right before the `exit 0` line:

```
cd /home/pi && ./px4 -d px4.config > px4.log
```

Parrot Bebop

Support for the Bebop is really early stage and should be used very carefully.

Build it

```
cd Firmware  
make posix_bebop_default
```

Turn on your Bebop and connect your host machine with the Bebop's wifi. Then, press the power button four times to enable ADB and to start the telnet daemon.

```
make posix_bebop_default upload
```

This will upload the PX4 mainapp into /usr/bin and create the file /home/root/parameters if not already present. In addition, we need the Bebop's mixer file and the px4.config.

Currently, both files have to be copied manually using the following commands.

```
adb connect 192.168.42.1:9050  
adb push ROMFS/px4fmu_common/mixers/bebop.main.mix /home/root  
adb push posix-configs/bebop/px4.config /home/root  
adb disconnect
```

Run it

Connect to the Bebop's wifi and press the power button four times. Next, connect with the Bebop via telnet or adb shell and run the commands bellow.

```
telnet 192.168.42.1
```

Kill the Bebop's proprietary driver with

```
kk
```

and start the PX4 mainapp with:

```
px4 /home/root/px4.config
```

In order to fly the Bebop, connect a joystick device with your host machine and start QGroundControl. Both, the Bebop and the joystick should be recognized. Follow the instructions to calibrate the sensors and setup your joystick device.

QuRT / Snapdragon based boards

Build it

The commands below build the targets for the Linux and the DSP side. Both executables communicate via [muORB](#).

```
cd Firmware  
make eagle_default
```

To load the SW on the device, connect via USB cable and make sure the device is booted. Run this in a new terminal window:

```
adb shell
```

Go back to previous terminal and upload:

```
make eagle_default upload
```

Note that this will also copy (and overwrite) the two config files [mainapp.config](#) and [px4.config](#) to the device. Those files are stored under `/usr/share/data/adsp/px4.config` and `/home/linaro/mainapp.config` respectively if you want to edit the startup scripts directly on your vehicle.

The mixer currently needs to be copied manually:

```
adb push ROMFS/px4fmu_common/mixers/quad_x.main.mix /usr/share/data/adsp
```

Run it

Run the DSP debug monitor:

```
 ${HEXAGON_SDK_ROOT}/tools/debug/mini-dm/Linux_Debug/mini-dm
```

Note: alternatively, especially on Mac, you can also use [nano-dm](#).

Go back to ADB shell and run px4:

```
cd /home/linaro  
.px4 mainapp.config
```

Note that the px4 will stop as soon as you disconnect the USB cable (or if your ssh session is disconnected). To fly, you should make the px4 auto-start after boot.

Autostart

To run the px4 as soon as the Snapdragon has booted, you can add the startup to

```
rc.local :
```

Either edit the file `/etc/rc.local` directly on the Snapdragon:

```
adb shell  
vim /etc/rc.local
```

Or copy the file to your computer, edit it locally, and copy it back:

```
adb pull /etc/rc.local  
gedit rc.local  
adb push rc.local /etc/rc.local
```

For the auto-start, add the following line before `exit 0`:

```
(cd /home/linaro && ./px4 mainapp.config > mainapp.log)  
  
exit 0
```

Make sure that the `rc.local` is executable:

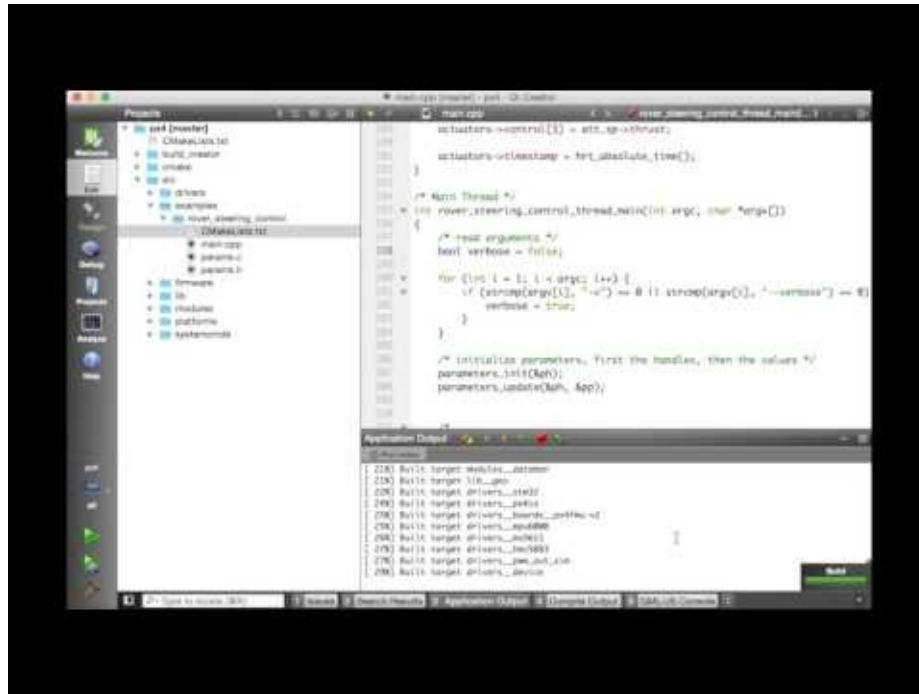
```
adb shell  
chmod +x /etc/rc.local
```

Then reboot the Snapdragon:

```
adb reboot
```

Compiling in a graphical IDE

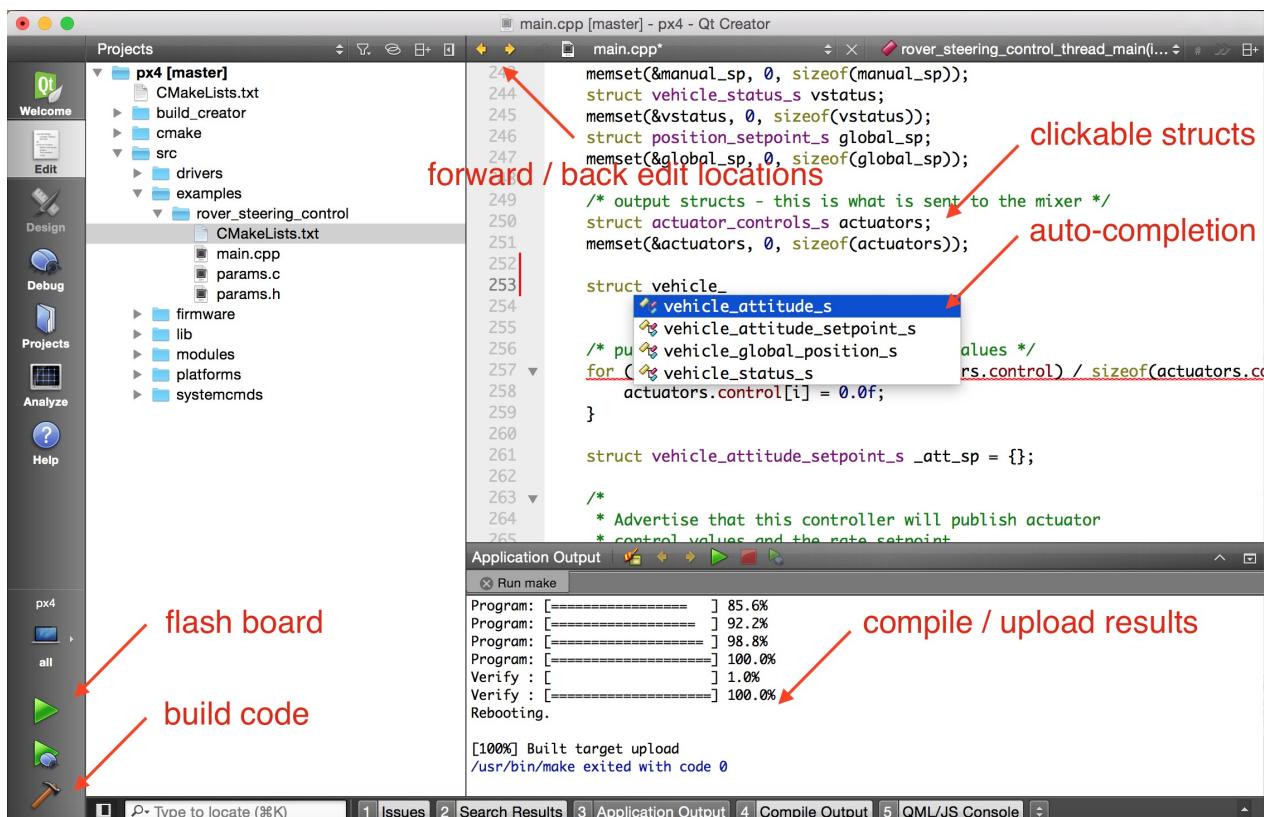
The PX4 system supports Qt Creator, Eclipse and Sublime Text. Qt Creator is the most user-friendly variant and hence the only officially supported IDE. Unless an expert in Eclipse or Sublime, their use is discouraged. Hardcore users can find an [Eclipse project](#) and a [Sublime project](#) in the source tree.



[Video link](#)

Qt Creator Functionality

Qt creator offers clickable symbols, auto-completion of the complete codebase and building and flashing firmware.



Qt Creator on Linux

Before starting Qt Creator, the [project file](#) needs to be created:

```
cd ~/src/Firmware
mkdir ../Firmware-build
cd ../Firmware-build
cmake ../Firmware -G "CodeBlocks - Unix Makefiles"
```

Then load the CMakeLists.txt in the root firmware folder via File -> Open File or Project -> Select the CMakeLists.txt file.

After loading, the 'play' button can be configured to run the project by selecting 'custom executable' in the run target configuration and entering 'make' as executable and 'upload' as argument.

Qt Creator on Windows

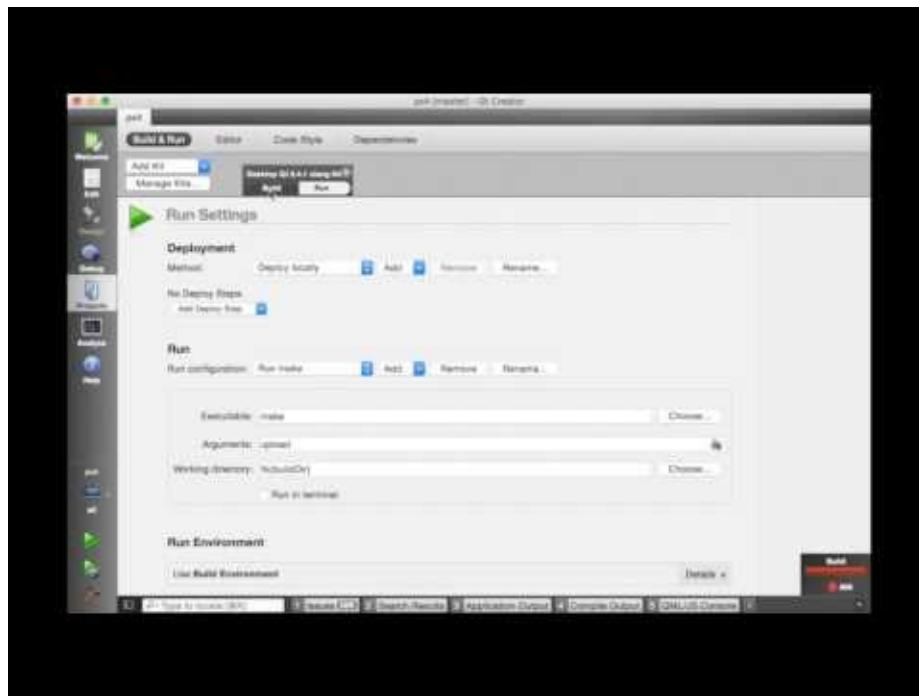
Windows has not been tested with Qt creator yet.

Qt Creator on Mac OS

Before starting Qt Creator, the [project file](#) needs to be created:

```
cd ~/src/Firmware  
mkdir build_creator  
cd build_creator  
cmake .. -G "CodeBlocks - Unix Makefiles"
```

That's it! Start Qt Creator, then complete the steps in the video below to set up the project to build.



[Video link](#)

Contributing

Contact information for the core dev team and community can be found below. The PX4 project uses a three-branch Git branching model:

- [master](#) is by default unstable and sees rapid development.
- [beta](#) has been thoroughly tested. It's intended for flight testers.
- [stable](#) points to the last release.

We try to retain a [linear history through rebases](#) and avoid the [Github flow](#). However, due to the global team and fast moving development we might resort to merges at times.

To contribute new functionality, [sign up for Github](#), then [fork](#) the repository, [create a new branch](#), add your changes, and finally [send a pull request](#). Changes will be merged when they pass our [continuous integration](#) tests.

All contributions have to be under the permissive [BSD 3-clause license](#) and all code must not impose any further constraints on the use.

Commits and Commit Messages

Please use descriptive, multi-paragraph commit messages for all non-trivial changes. Structure them well so they make sense in the one-line summary but also provide full detail.

Component: Explain the change in one sentence. Fixes #1234

Prepend the software component to the start of the summary line, either by the module name or a description of it. (e.g. "mc_att_ctrl" or "multicopter attitude controller").

If the issue number is appended as <Fixes #1234>, Github will automatically close the issue when the commit is merged to the master branch.

The body of the message can contain several paragraphs. Describe in detail what you changed. Link issues and flight logs either related to this fix or to the testing results of this commit.

Describe the change and why you changed it, avoid to paraphrase the code change (Good: "Adds an additional safety check for vehicles with low quality GPS reception". Bad: "Add gps_reception_check() function").

Reported-by: Name <email@px4.io>

Use `git commit -s` to sign off on all of your commits. This will add `signed-off-by:` with your name and email as the last line.

This commit guide is based on best practices for the Linux Kernel and other [projects maintained](#) by Linus Torvalds.

Tests Flight Results

Test flights are important for quality assurance. Please upload the logs from the microSD card to [Log Muncher](#) and share the link on the [PX4 Discuss](#) along with a verbal flight report.

Forums and Chat

- [Google+](#)
- [Gitter](#)
- [PX4 Discuss](#)

Weekly Dev Call

The PX4 Dev Team syncs up on its weekly dev call (connect via [Mumble](#) client).

- TIME: 19:00h Zurich time, 1 p.m. Eastern Time, 10 a.m. Pacific Standard Time
- Server: sitl01.dronetest.io
- Port: 64738
- Password: px4
- The agenda is announced in advance on the [PX4 Discuss](#)
- Issues and PRs may be labelled "devcall" to flag them for discussion

GIT Examples

Contributing code to PX4

Adding a feature to PX4 follows a defined workflow. In order to share your contributions on PX4, you can follow this example.

- [Sign up](#) for github if you haven't already
- Fork the Firmware (see [here](#))
- Clone your forked repository to your local computer

```
cd ~/wherever/
git clone https://github.com/<your git name>/Firmware.git
```

- Go into the new directory, initialize and update the submodules, and add the original upstream Firmware

```
cd Firmware
git submodule update --init --recursive
git remote add upstream https://github.com/PX4/Firmware.git
```

- You should have now two remote repositories: One repository is called upstream that points to the PX4 Firmware, and one repository that points to your forked repository of the PX4 repository.
- This can be checked with the following command:

```
git remote -v
```

- Make the changes that you want to add to the current master.
- Create a new branch with a meaningful name that represents your feature

```
git checkout -b <your feature branch name>
```

you can use the command `git branch` to make sure you're on the right branch.

- Add your changes that you want to be part of the commit by adding the respective files

```
git add <file name>
```

If you prefer having a GUI to add your files see [Gitk](#) or `git add -p`.

- Commit the added files with a meaningful message explaining your changes

```
git commit -m "<your commit message>"
```

For a good commit message, please refer to [Contributing](#) section.

- Some time might have passed and the [upstream master](#) has changed. PX4 prefers a linear commit history and uses [git rebase](#). To include the newest changes from upstream in your local branch, switch to your master branch

```
git checkout master
```

Then pull the newest commits from upstream master

```
git pull upstream master
```

Now your local master is up to date. Switch back to your feature branch

```
git checkout <your feature branch name>
```

and rebase on your updated master

```
git rebase master
```

- Now you can push your local commits to your forked repository

```
git push origin <your feature branch name>
```

- You can verify that the push was successful by going to your forked repository in your browser: <https://github.com/<your git name>/Firmware.git>

There you should see the message that a new branch has been pushed to your forked repository.

- Now it's time to create a pull request (PR). On the right hand side of the "new branch message" (see one step before), you should see a green button saying "Compare & Create Pull Request". Then it should list your changes and you can (must) add a meaningful title (in case of a one commit PR, it's usually the commit message) and message ([explain what you did for what reason](#). Check [other pull requests](#) for comparison)
- You're done! Responsible members of PX4 will now have a look at your contribution and decide if they want to integrate it. Check if they have questions on your changes every once in a while.

Update Submodule

There are several ways to update a submodule. Either you clone the repository or you go in the submodule directory and follow the same procedure as in [Contributing code to PX4](#).

Do a PR for a submodule update

This is required after you have done a PR for a submodule X repository and the bug-fix / feature-add is in the current master of submodule X. Since the Firmware still points to a commit before your update, a submodule pull request is required such that the submodule used by the Firmware points to the newest commit.

```
cd Firmware
```

- Make a new branch that describes the fix / feature for the submodule update:

```
git checkout -b pr-some-fix
```

- Go to submodule subdirectory

```
cd <path to submodule>
```

- PX4 submodule might not necessarily point to the newest commit. Therefore, first checkout master and pull the newest upstream code.

```
git checkout master  
git pull origin master
```

- Go back to Firmware directory, and as usual add, commit and push the changes.

```
cd -  
git add <path to submodule>  
git commit -m "Update submodule to include ..."  
git push upstream pr-some-fix
```

Checkout pull requests

You can test someone's pull request (changes are not yet merged) even if the branch to merge only exists on the fork from that person. Do the following

```
git fetch upstream pull/<PR ID>/head:<branch name>
```

`PR ID` is the number right next to the PR's title (without the #) and the `<branch name>` can also be found right below the `PR ID`, e.g. `<the other persons git name>:<branch name>`. After that you can see the newly created branch locally with

```
git branch
```

Then switch to that branch

```
git checkout <branch name>
```

Common pitfalls

Force push to forked repository

After having done the first PR, people from the PX4 community will review your changes. In most cases this means that you have to fix your local branch according to the review. After changing the files locally, the feature branch needs to be rebased again with the most recent upstream/master. However, after the rebase, it is no longer possible to push the feature branch to your forked repository directly, but instead you need to use a force push:

```
git push --force-with-lease origin <your feature branch name>
```

Rebase merge conflicts

If a conflict occurs during a `git rebase`, please refer to [this guide](#).

Pull merge conflicts

If a conflict occurs during a `git pull`, please refer to [this guide](#).

Flight Modes

Flight Modes define the state of the system at any given time. The user transitions between flight modes via switches on the remote control or the [ground control station](#).

Flight Mode Quick Summary

- **MANUAL**

- **Fixed wing aircraft/ rovers / boats:**
 - **MANUAL:** The pilot's control inputs are passed directly to the output mixer.
 - **STABILIZED:** The pilot's inputs are passed as roll and pitch *angle* commands and a manual yaw command.
- **Multicopters:**
 - **ACRO:** The pilot's inputs are passed as roll, pitch, and yaw *rate* commands to the autopilot. This allows the multicopter to become completely inverted. Throttle is passed directly to the output mixer
 - **RATTITUDE** The pilot's inputs are passed as roll, pitch, and yaw *rate* commands to the autopilot if they are greater than the mode's threshold. If not the inputs are passed as roll and pitch *angle* commands and a yaw *rate* command. Throttle is passed directly to the output mixer.
 - **ANGLE** The pilot's inputs are passed as roll and pitch *angle* commands and a yaw *rate* command. Throttle is passed directly to the output mixer.

- **ASSISTED**

- **ALTCTL**

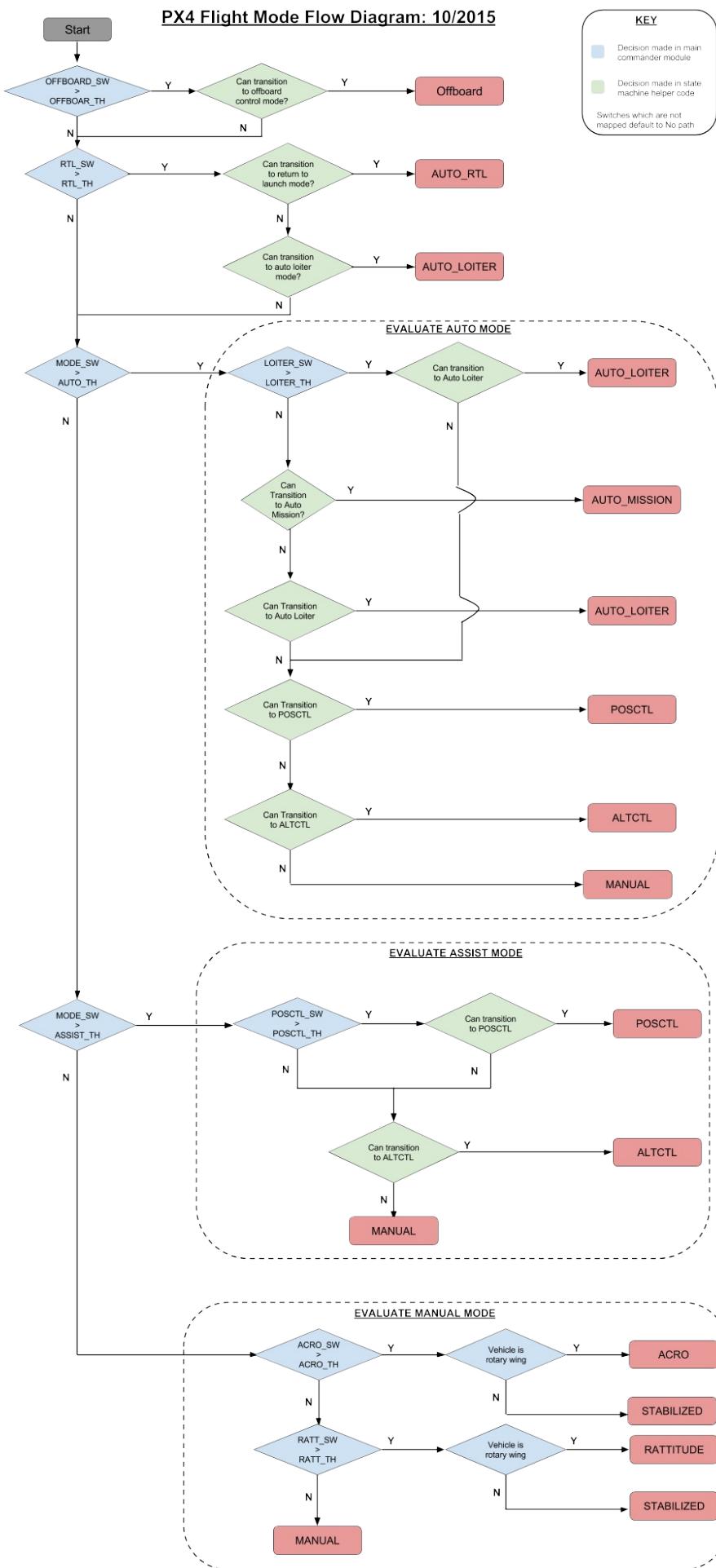
- **Fixed wing aircraft:** When the roll, pitch and yaw inputs (RPY) are all centered (less than some specified deadband range) the aircraft will return to straight and level flight and keep its current altitude. It will drift with the wind.
- **Multicopters:** Roll, pitch and yaw inputs are as in MANUAL mode. Throttle inputs indicate climb or sink at a predetermined maximum rate. Throttle has large deadzone.

- **POSCTL**

- **Fixed wing aircraft:** Neutral inputs give level, flight and it will crab against the wind if needed to maintain a straight line.
- **Multicopters** Roll controls left-right speed, pitch controls front-back speed over ground. When roll and pitch are all centered (inside deadzone) the multicopter will hold position. Yaw controls yaw rate as in MANUAL mode. Throttle controls climb/descent rate as in ALTCTL mode.

- **AUTO**
 - **AUTO_LOITER**
 - **Fixed wing aircraft:** The aircraft loiters around the current position at the current altitude (or possibly slightly above the current altitude, good for 'I'm losing it').
 - **Multirotors:** The multirotor hovers / loiters at the current position and altitude.
 - **AUTO_RTL**
 - **Fixed wing aircraft:** The aircraft returns to the home position and loiters in a circle above the home position.
 - **Multirotors:** The multirotor returns in a straight line on the current altitude (if higher than the home position + loiter altitude) or on the loiter altitude (if higher than the current altitude), then lands automatically.
 - **AUTO_MISSION**
 - **All system types:** The aircraft obeys the programmed mission sent by the ground control station (GCS). If no mission received, aircraft will LOITER at current position instead.
- **OFFBOARD** In this mode the position, velocity or attitude reference / target / setpoint is provided by a companion computer connected via serial cable and MAVLink. The offboard setpoint can be provided by APIs like [MAVROS](#) or [Dronekit](#).

Flight Mode Evaluation Diagram



Architectural Overview

PX4 consists of two main layers: The [PX4 flight stack](#), an autopilot software solution and the [PX4 middleware](#), a general robotics middleware which can support any type of autonomous robot.

All [airframes](#), and in fact all robotic systems including boats, share a single codebase. The complete system design is [reactive](#), which means that:

- All functionality is divided into exchangable components
- Communication is done by asynchronous message passing
- The system can deal with varying workload

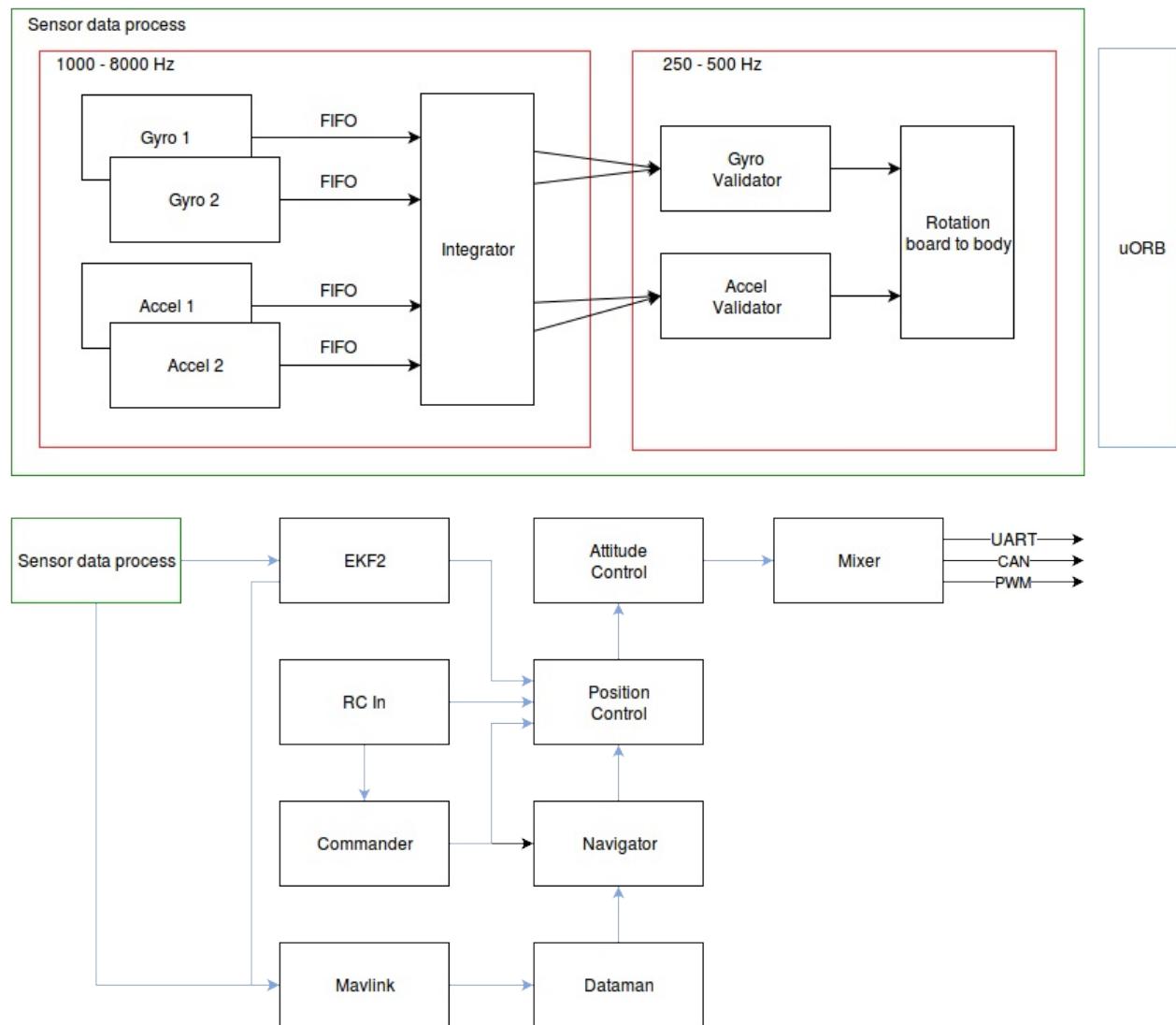
In addition to these runtime considerations, its modularity maximizes [reusability](#).

High Level Software Architecture

Each of the blocks below is a separate module, which is self-contained in terms of code, dependencies and even at runtime. Each arrow is a connection through publish/subscribe calls through [uORB](#).

Info The architecture of PX4 allows to exchange every single of these blocks very rapidly and conveniently, even at runtime.

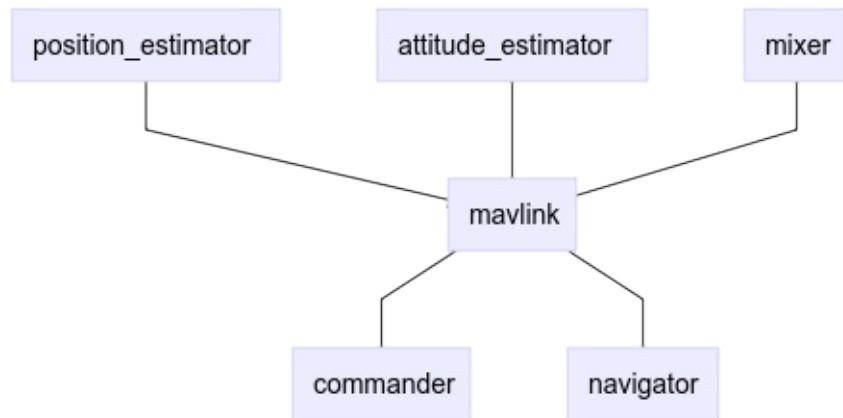
The controllers / mixers are specific to a particular airframe (e.g. a multicopter, VTOL or plane), but the higher-level mission management blocks like the `commander` and `navigator` are shared between platforms.



Info This flow chart can be updated from [here](#) and open it with draw.io Diagrams.

Communication Architecture with the GCS

The interaction with the ground control station (GCS) is handled through the "business logic" applications including the commander (general command & control, e.g. arming), the navigator (accepts missions and turns them into lower-level navigation primitives) and the mavlink application, which accepts MAVLink packets and converts them into the onboard uORB data structures. This isolation has been architected explicitly to avoid having a MAVLink dependency deep in the system. The MAVLink application also consumes a lot of sensor data and state estimates and sends them to the ground control station.

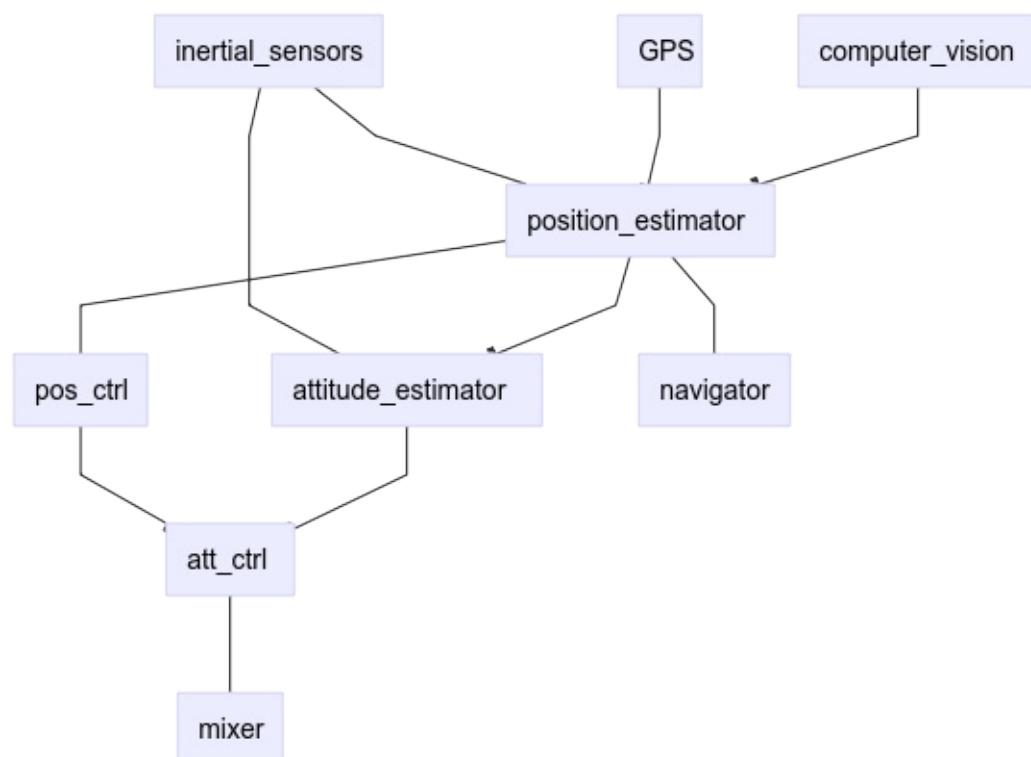


PX4 Flight Stack

The PX4 flight stack is a collection of guidance, navigation and control algorithms for autonomous drones. It includes controllers for fixed wing, multirotor and VTOL airframes as well as estimators for attitude and position.

Estimation and Control Architecture

The diagram below shows an example implementation of the typical blocks. Depending on the vehicle some of these can be also combined into a single application (e.g. when a model predictive controller for a specific vehicle is wanted).



PX4 Middleware

The PX4 Middleware consists primarily of device drivers for embedded sensors and a publish-subscribe based middleware to connect these sensors to applications running the [flight controls](#).

The use of the publish-subscribe scheme means that:

- The system is reactive: It will update instantly when new data is available
- It is running fully parallelized
- A system component can consume data from anywhere in a thread-safe fashion

Mixing and Actuators

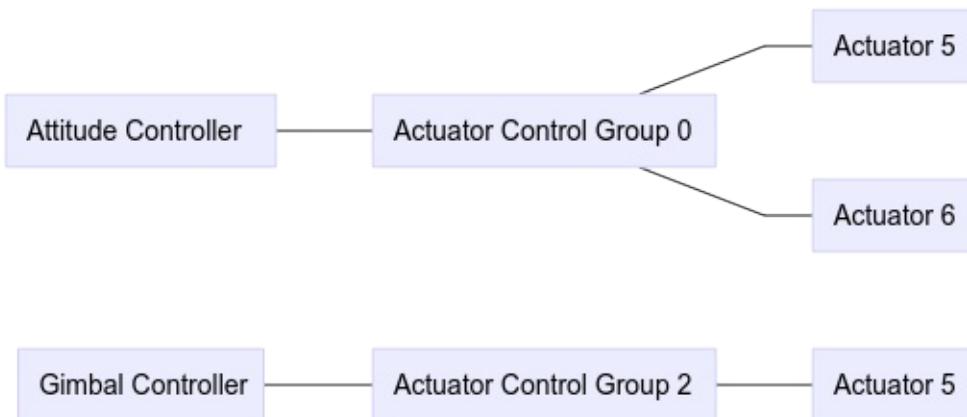
The PX4 architecture ensures that the airframe layout does not require special case handling in the core controllers.

Mixing means to take force commands (e.g. `turn right`) and translate them to actuator commands which control motors or servos. For a plane with one servo per aileron this means to command one of them high and the other low. The same applies for multicopters: Pitching forward requires changing the speed of all motors.

Separating the mixer logic from the actual attitude controller greatly improves reusability.

Control Pipeline

A particular controller sends a particular normalized force or torque demand (scaled from -1..+1) to the mixer, which then sets individual actuators accordingly. The output driver (e.g. UART, UAVCAN or PWM) then scales it to the actuators native units, e.g. a PWM value of 1300.



Control Groups

PX4 uses control groups (inputs) and output groups. Conceptually they are very simple: A control group is e.g. `attitude`, for the core flight controls, or `gimbal` for payload. An output group is one physical bus, e.g. the first 8 PWM outputs for servos. Each of these groups has 8 normalized (-1..+1) command ports, which can be mapped and scaled through the mixer. A mixer defines how each of these 8 signals of the controls are connected to the 8 outputs.

For a simple plane control 0 (roll) is connected straight to output 0 (aileron). For a multicopter things are a bit different: control 0 (roll) is connected to all four motors and combined with throttle.

Control Group #0 (Flight Control)

- 0: roll (-1..1)
- 1: pitch (-1..1)
- 2: yaw (-1..1)
- 3: throttle (0..1 normal range, -1..1 for variable pitch / thrust reversers)
- 4: flaps (-1..1)
- 5: spoilers (-1..1)
- 6: airbrakes (-1..1)
- 7: landing gear (-1..1)

Control Group #1 (Flight Control VTOL/Alternate)

- 0: roll ALT (-1..1)
- 1: pitch ALT (-1..1)
- 2: yaw ALT (-1..1)
- 3: throttle ALT (0..1 normal range, -1..1 for variable pitch / thrust reversers)
- 4: reserved / aux0
- 5: reserved / aux1
- 6: reserved / aux2
- 7: reserved / aux3

Control Group #2 (Gimbal)

- 0: gimbal roll
- 1: gimbal pitch
- 2: gimbal yaw
- 3: gimbal shutter
- 4: reserved
- 5: reserved
- 6: reserved
- 7: reserved (parachute, -1..1)

Control Group #3 (Manual Passthrough)

- 0: RC roll
- 1: RC pitch

- 2: RC yaw
- 3: RC throttle
- 4: RC mode switch
- 5: RC aux1
- 6: RC aux2
- 7: RC aux3

Control Group #6 (First Payload)

- 0: function 0 (default: parachute)
- 1: function 1
- 2: function 2
- 3: function 3
- 4: function 4
- 5: function 5
- 6: function 6
- 7: function 7

Virtual Control Groups

These groups are NOT mixer inputs, but serve as meta-channels to feed fixed wing and multicopter controller outputs into the VTOL governor module.

Control Group #4 (Flight Control MC VIRTUAL)

- 0: roll ALT (-1..1)
- 1: pitch ALT (-1..1)
- 2: yaw ALT (-1..1)
- 3: throttle ALT (0..1 normal range, -1..1 for variable pitch / thrust reversers)
- 4: reserved / aux0
- 5: reserved / aux1
- 6: reserved / aux2
- 7: reserved / aux3

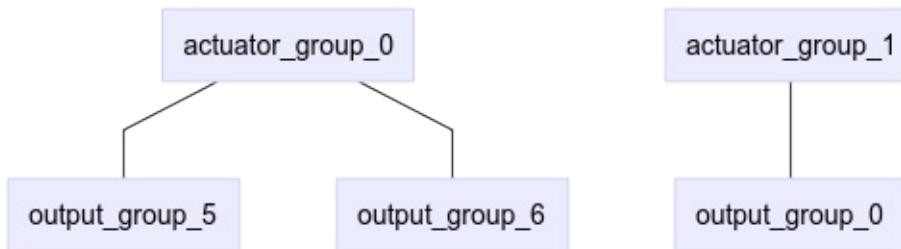
Control Group #5 (Flight Control FW VIRTUAL)

- 0: roll ALT (-1..1)
- 1: pitch ALT (-1..1)
- 2: yaw ALT (-1..1)
- 3: throttle ALT (0..1 normal range, -1..1 for variable pitch / thrust reversers)
- 4: reserved / aux0

- 5: reserved / aux1
- 6: reserved / aux2
- 7: reserved / aux3

Mapping

Since there are multiple control groups (like flight controls, payload, etc.) and multiple output groups (first 8 PWM outputs, UAVCAN, etc.), one control group can send command to multiple output groups.



PX4 mixer definitions

Files in `ROMFS/px4fmu_common/mixers` implement mixers that are used for predefined airframes. They can be used as a basis for customisation, or for general testing purposes.

Syntax

Mixer definitions are text files; lines beginning with a single capital letter followed by a colon are significant. All other lines are ignored, meaning that explanatory text can be freely mixed with the definitions.

Each file may define more than one mixer; the allocation of mixers to actuators is specific to the device reading the mixer definition, and the number of actuator outputs generated by a mixer is specific to the mixer.

For example: each simple or null mixer is assigned to outputs 1 to x in the order they appear in the mixer file.

A mixer begins with a line of the form

```
<tag>: <mixer arguments>
```

The tag selects the mixer type; 'M' for a simple summing mixer, 'R' for a multirotor mixer, etc.

Null Mixer

A null mixer consumes no controls and generates a single actuator output whose value is always zero. Typically a null mixer is used as a placeholder in a collection of mixers in order to achieve a specific pattern of actuator outputs.

The null mixer definition has the form:

```
Z:
```

Simple Mixer

A simple mixer combines zero or more control inputs into a single actuator output. Inputs are scaled, and the mixing function sums the result before applying an output scaler.

A simple mixer definition begins with:

```
M: <control count>
0: <-ve scale> <+ve scale> <offset> <lower limit> <upper limit>
```

If `<control count>` is zero, the sum is effectively zero and the mixer will output a fixed value that is `<offset>` constrained by `<lower limit>` and `<upper limit>`.

The second line defines the output scaler with scaler parameters as discussed above. Whilst the calculations are performed as floating-point operations, the values stored in the definition file are scaled by a factor of 10000; i.e. an offset of -0.5 is encoded as -5000.

The definition continues with `<control count>` entries describing the control inputs and their scaling, in the form:

```
S: <group> <index> <-ve scale> <+ve scale> <offset> <lower limit> <upper limit>
```

The `<group>` value identifies the control group from which the scaler will read, and the `<index>` value an offset within that group. These values are specific to the device reading the mixer definition.

When used to mix vehicle controls, mixer group zero is the vehicle attitude control group, and index values zero through three are normally roll, pitch, yaw and thrust respectively.

The remaining fields on the line configure the control scaler with parameters as discussed above. Whilst the calculations are performed as floating-point operations, the values stored in the definition file are scaled by a factor of 10000; i.e. an offset of -0.5 is encoded as -5000.

Multirotor Mixer

The multirotor mixer combines four control inputs (roll, pitch, yaw, thrust) into a set of actuator outputs intended to drive motor speed controllers.

The mixer definition is a single line of the form:

```
R: <geometry> <roll scale> <pitch scale> <yaw scale> <deadband>
```

The supported geometries include:

- 4x - quadrotor in X configuration
- 4+ - quadrotor in + configuration
- 6x - hexcopter in X configuration
- 6+ - hexcopter in + configuration
- 8x - octocopter in X configuration
- 8+ - octocopter in + configuration

Each of the roll, pitch and yaw scale values determine scaling of the roll, pitch and yaw controls relative to the thrust control. Whilst the calculations are performed as floating-point operations, the values stored in the definition file are scaled by a factor of 10000; i.e. an factor of 0.5 is encoded as 5000.

Roll, pitch and yaw inputs are expected to range from -1.0 to 1.0, whilst the thrust input ranges from 0.0 to 1.0. Output for each actuator is in the range -1.0 to 1.0.

In the case where an actuator saturates, all actuator values are rescaled so that the saturating actuator is limited to 1.0.

PWM_limit State Machine

[PWM_limit State Machine] Controls PWM outputs as a function of pre-armed and armed inputs. Provides a delay between assertion of "armed" and a ramp-up of throttle on assertion of the armed signal.

Quick Summary

Inputs

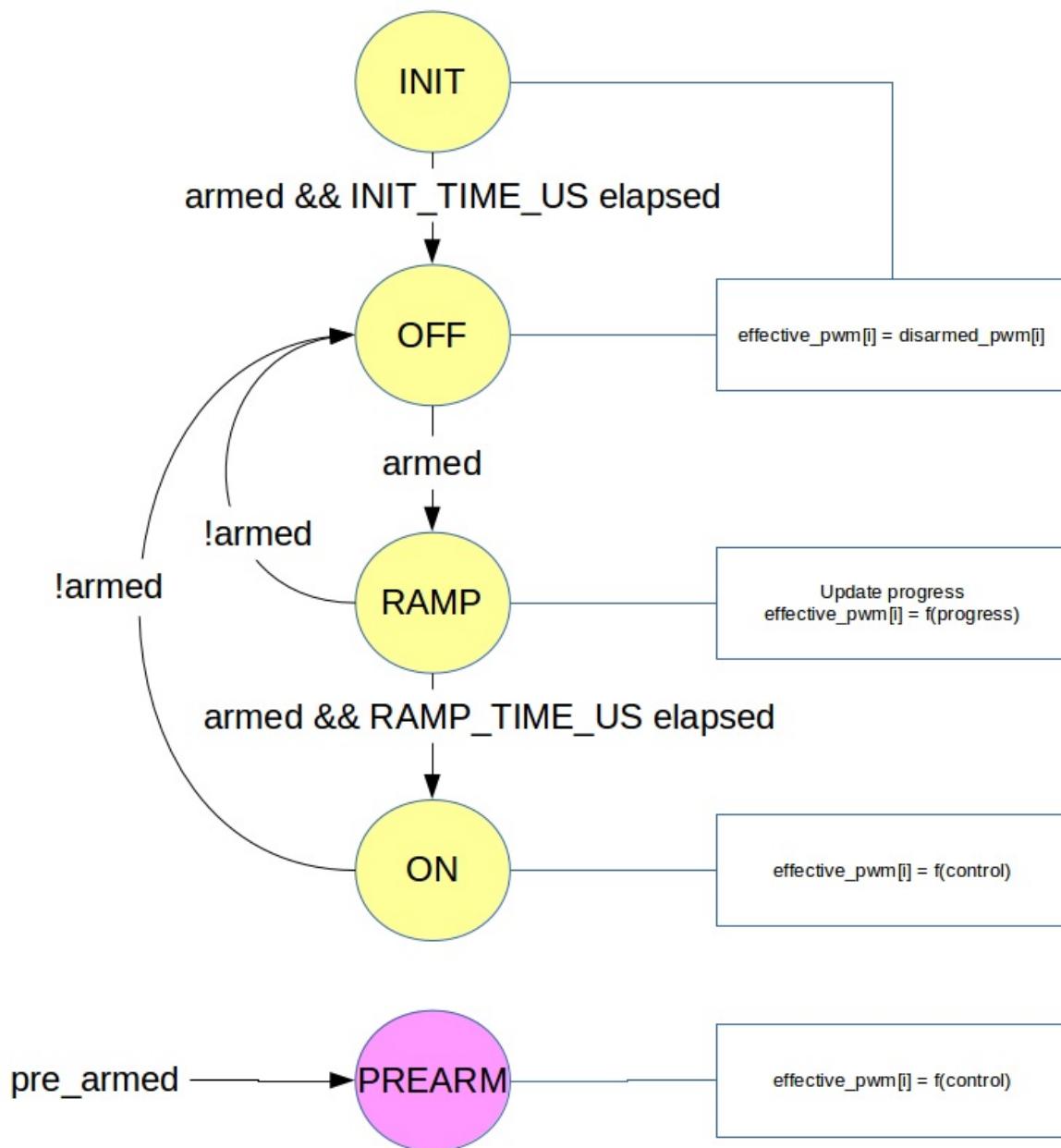
- armed: asserted to enable dangerous behaviors such as spinning propellers
- pre-armed: asserted to enable benign behaviors such as moving control surfaces
 - this input overrides the current state
 - assertion of pre-armed immediately forces behavior of state ON, regardless of current state ** deassertion of pre-armed reverts behavior to current state

States

- INIT and OFF
 - pwm outputs set to disarmed values.
- RAMP
 - pwm outputs ramp from disarmed values to min values.
- ON
 - pwm outputs set according to control values.

State Transition Diagram

PWM_LIMIT_STATE_



Note: input pre_armed overrides the current state while asserted. When deasserted the state machine functions normally. Throttle controls must be set to NaN while pre_armed is asserted.

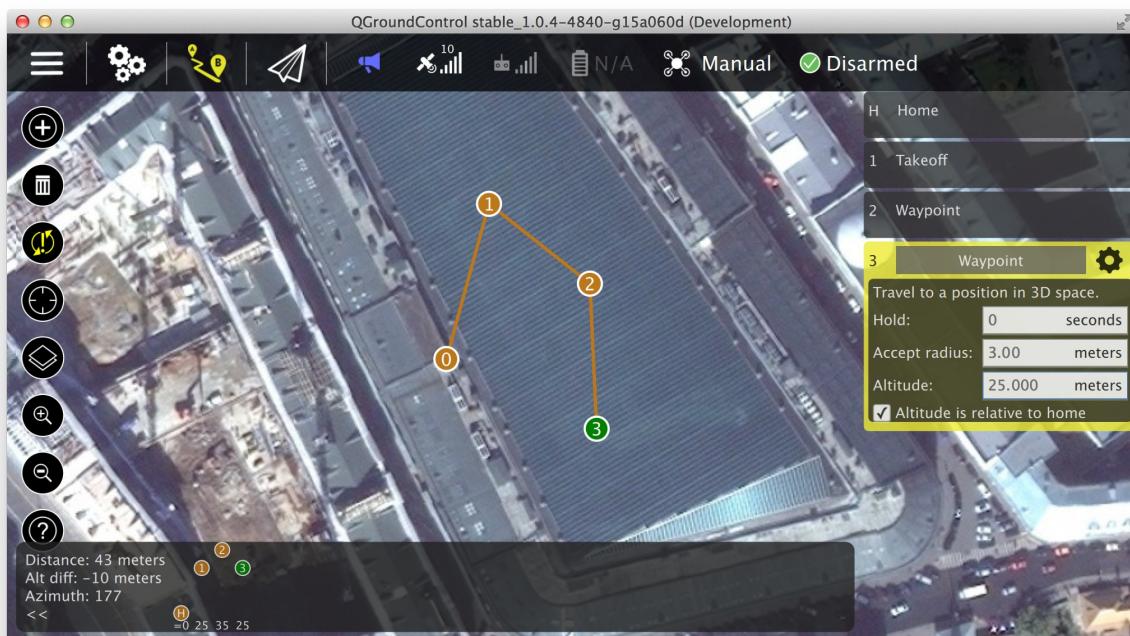
QGroundControl

QGroundControl is an app to configure and fly a PX4 based autopilot. It is cross platform and supports all major operating systems:

- Mobile: Android and iOS (currently focused on tablet)
- Desktop: Windows, Linux, Mac OS

Planning Missions

To plan a new mission, switch to the planning tab, click on the + icon in the top left and click on the map to create waypoints. A context menu will open on the side to adjust the waypoints. Click on the highlight transmission icon to send them to the vehicle.



Flying Missions

Switch to the flying tab. The mission should be visible on the map. Click on the current flight mode to change it to MISSION and click on DISARMED to arm the vehicle. If the vehicle is already in flight it will fly to the first leg of the mission and then follow it.



Setting parameters

Switch to the setup tab. Scroll the menu on the left all the way to the bottom and click on the parameter icon. Parameters can be changed by double-clicking on them, which opens a context menu to edit, along with a more detailed description.

QGroundControl stable_1.0.4-4840-g15a060d (Development)			
Groups	Multicopter Attitude Control Parameters		
Summary	Camera trigger	MC_ACRO_P_MAX	360.000 deg/s Max acro pitch rate
Firmware	Circuit Breaker	MC_ACRO_R_MAX	360.000 deg/s Max acro roll rate
Airframe	Commander	MC_ACRO_Y_MAX	360.000 deg/s Max acro yaw rate
Radio	Data Link Loss	MC_PITCHRATE_D	0.003 Pitch rate D gain
Flight Modes	GPS Failure Navigation	MC_PITCHRATE_FF	0.000 Pitch rate feedforward
Sensors	Geofence	MC_PITCHRATE_I	0.050 Pitch rate I gain
Power	Land Detector	MC_PITCHRATE_MAX	220.000 deg/s Max pitch rate
Safety	MAVLink	MC_PITCHRATE_P	0.150 Pitch rate P gain
Parameters	Miscellaneous	MC_PITCH_P	7.000 1/s Pitch P gain
	Mission	MC_RATT_TH	1.000 Threshold for Rattitude mode
Multicopter Attitude Control			
Multicopter Position Control			
PWM Outputs			
Position Estimator INAV			
Radio Calibration			
Radio Signal Loss			

Installation

QGroundControl can be downloaded from its [website](#).

Developers are advised to use the latest daily build instead of the stable release.

Building from source

Firmware developers are encouraged to build from source in order to have a matching recent version to their flight code.

Follow the [QGroundControl build instructions](#) to install Qt and build the source code.

First App Tutorial (Hello Sky)

This tutorial explains in detail how to create a new onboard application and how to run it.

Prerequisites

- Pixhawk or Snapdragon compatible autopilot
- PX4 Toolchain [installed](#)
- Github Account ([sign up for free](#))

Step 1: File Setup

To conveniently manage your custom code and pull in updates from the main repository, it is recommended to fork the Firmware repository with the GIT version control system:

- [Sign up](#) for Github
- Go to the [Firmware repository website](#) and click **FORK** on the upper right part.
- If you are not already there, open the website of your fork and copy the private repository URL in the center.
- Clone the repository to your hard drive, e.g. on the command line via `git clone https://github.com/<youraccountname>/Firmware.git`. Windows users please [refer to the Github help](#) and e.g. fork / clone with their Github for Windows app.
- Update the git submodules: Run in your shell (on Windows in the PX4 console).

```
cd Firmware  
git submodule init  
git submodule update --recursive
```

Enter the `Firmware/src/examples/` directory on your local hard drive and look at the files in the directory.

Step 2: Minimal Application

Create a new C file named `px4_simple_app.c` in the `px4_simple_app` folder (it will already be present, delete the existing file for the maximum educational effect).

Edit it and start with the default header and a main function.

Note the code style in this file - all contributions to PX4 should adhere to it.

```
*****
*
*   Copyright (c) 2012-2016 PX4 Development Team. All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in
*    the documentation and/or other materials provided with the
*    distribution.
* 3. Neither the name PX4 nor the names of its contributors may be
*    used to endorse or promote products derived from this software
*    without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
* "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
* LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
* FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
* COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
* INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
* BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
* OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
* AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
* ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*
***** /
```



```
/**
* @file px4_simple_app.c
* Minimal application example for PX4 autopilot
*
* @author Example User <mail@example.com>
*/
```



```
#include <px4_config.h>
#include <px4_tasks.h>
#include <px4_posix.h>
#include <unistd.h>
#include <stdio.h>
#include <poll.h>
#include <string.h>

#include <uORB/uORB.h>
#include <uORB/topics/sensor_combined.h>
#include <uORB/topics/vehicle_attitude.h>
```

```
__EXPORT int px4_simple_app_main(int argc, char *argv[]);

int px4_simple_app_main(int argc, char *argv[])
{
    PX4_INFO("Hello Sky!");
    return OK;
}
```

Step 3: Register the Application in NuttShell and build it

The application is now complete and could be run, but it is not registered as NuttShell command yet. To enable the compilation of the application into the firmware, add it to the list of modules to build, which is here:

- Posix SITL: [Firmware/cmake/configs posix_sitl_default.cmake](#)
- Pixhawk v1/2: [Firmware/cmake/configs nuttx_px4fmu-v2_default.cmake](#)
- Pixracer: [Firmware/cmake/configs nuttx_px4fmu-v4_default.cmake](#)

Create a new line for your application somewhere in the file:

```
examples/px4_simple_app
```

Build it:

- Pixhawk v1/2: `make px4fmu-v2_default`
- Pixhawk v3: `make px4fmu-v4_default`

Step 4: Upload and Test the app

Enable the uploader and then reset the board:

- Pixhawk v1/2: `make px4fmu-v2_default upload`
- Pixhawk v3: `make px4fmu-v4_default upload`

It should print before you reset the board a number of compile messages and at the end:

```
Loaded firmware for X,X, waiting for the bootloader...
```

Once the board is reset, and uploads, it prints:

```
Erase : [=====] 100.0%
Program: [=====] 100.0%
Verify : [=====] 100.0%
Rebooting.

[100%] Built target upload
```

Connect the console

Now connect to the [system console](#) either via serial or USB. Hitting ENTER will bring up the shell prompt:

```
nsh>
```

Type "help" and hit ENTER

```
nsh> help
help usage: help [-v] [<cmd>]

[      df      kill      mkfifo      ps      sleep
?      echo      losetup    mkrd      pwd      test
cat    exec      ls        mh        rm       umount
cd     exit      mb        mount    rmdir    unset
cp     free      mkdir     mv        set      usleep
dd     help      mkfatfs  mw        sh       xd

Builtin Apps:
reboot
perf
top
...
px4_simple_app
...
sercon
serdis
```

Note that `px4_simple_app` is now part of the available commands. Start it by typing `px4_simple_app` and ENTER:

```
nsh> px4_simple_app
Hello Sky!
```

The application is now correctly registered with the system and can be extended to actually perform useful tasks.

Step 5: Subscribing Sensor Data

To do something useful, the application needs to subscribe inputs and publish outputs (e.g. motor or servo commands). Note that the *true* hardware abstraction of the PX4 platform comes into play here -- no need to interact in any way with sensor drivers and no need to update your app if the board or sensors are updated.

Individual message channels between applications are called *topics* in PX4. For this tutorial, we are interested in the [sensor_combined topic](#), which holds the synchronized sensor data of the complete system.

Subscribing to a topic is swift and clean:

```
#include <uORB/topics/sensor_combined.h>
...
int sensor_sub_fd = orb_subscribe(ORB_ID(sensor_combined));
```

The `sensor_sub_fd` is a topic handle and can be used to very efficiently perform a blocking wait for new data. The current thread goes to sleep and is woken up automatically by the scheduler once new data is available, not consuming any CPU cycles while waiting. To do this, we use the [poll\(\)](#) POSIX system call.

Adding `poll()` to the subscription looks like (*pseudocode, look for the full implementation below*):

```
#include <poll.h>
#include <uORB/topics/sensor_combined.h>
...
int sensor_sub_fd = orb_subscribe(ORB_ID(sensor_combined));

/* one could wait for multiple topics with this technique, just using one here */
px4_pollfd_struct_t fds[] = {
    { .fd = sensor_sub_fd, .events = POLLIN },
};

while (true) {
    /* wait for sensor update of 1 file descriptor for 1000 ms (1 second) */
    int poll_ret = px4_poll(fds, 1, 1000);
    ...

    if (fds[0].revents & POLLIN) {
        /* obtained data for the first file descriptor */
        struct sensor_combined_s raw;
        /* copy sensors raw data into local buffer */
        orb_copy(ORB_ID(sensor_combined), sensor_sub_fd, &raw);
        PX4_INFO("Accelerometer:\t%8.4f\t%8.4f\t%8.4f",
                 (double)raw.accelerometer_m_s2[0],
                 (double)raw.accelerometer_m_s2[1],
                 (double)raw.accelerometer_m_s2[2]);
    }
}
}
```

Compile the app now by issuing

```
make
```

Testing the uORB Subscription

The final step is to start your application as background application.

```
px4_simple_app &
```

Your app will flood the console with the current sensor values:

[px4_simple_app] Accelerometer:	0.0483	0.0821	0.0332
[px4_simple_app] Accelerometer:	0.0486	0.0820	0.0336
[px4_simple_app] Accelerometer:	0.0487	0.0819	0.0327
[px4_simple_app] Accelerometer:	0.0482	0.0818	0.0323
[px4_simple_app] Accelerometer:	0.0482	0.0827	0.0331
[px4_simple_app] Accelerometer:	0.0489	0.0804	0.0328

It will exit after printing five values. The next tutorial page will explain how to write a deamon which can be controlled from the commandline.

Step 7: Publishing Data

To use the calculated outputs, the next step is to *publish* the results. If we use a topic from which we know that the "mavlink" app forwards it to the ground control station, we can even look at the results. Let's hijack the attitude topic for this purpose.

The interface is pretty simple: Initialize the struct of the topic to be published and advertise the topic:

```
#include <uORB/topics/vehicle_attitude.h>
...
/* advertise attitude topic */
struct vehicle_attitude_s att;
memset(&att, 0, sizeof(att));
orb_advert_t att_pub_fd = orb_advertise(ORB_ID(vehicle_attitude), &att);
```

In the main loop, publish the information whenever its ready:

```
orb_publish(ORB_ID(vehicle_attitude), att_pub_fd, &att);
```

The modified complete example code is now:

```
#include <px4_config.h>
#include <px4_tasks.h>
#include <px4_posix.h>
#include <unistd.h>
#include <stdio.h>
#include <poll.h>
#include <string.h>

#include <uORB/uORB.h>
#include <uORB/topics/sensor_combined.h>
#include <uORB/topics/vehicle_attitude.h>

__EXPORT int px4_simple_app_main(int argc, char *argv[]);

int px4_simple_app_main(int argc, char *argv[])
{
    PX4_INFO("Hello Sky!");

    /* subscribe to sensor_combined topic */
    int sensor_sub_fd = orb_subscribe(ORB_ID(sensor_combined));
    /* limit the update rate to 5 Hz */
```

```

orb_set_interval(sensor_sub_fd, 200);

/* advertise attitude topic */
struct vehicle_attitude_s att;
memset(&att, 0, sizeof(att));
orb_advert_t att_pub = orb_advertise(ORB_ID(vehicle_attitude), &att);

/* one could wait for multiple topics with this technique, just using one here */
px4_pollfd_struct_t fds[] = {
    { .fd = sensor_sub_fd, .events = POLLIN },
    /* there could be more file descriptors here, in the form like:
     * { .fd = other_sub_fd, .events = POLLIN },
     */
};

int error_counter = 0;

for (int i = 0; i < 5; i++) {
    /* wait for sensor update of 1 file descriptor for 1000 ms (1 second) */
    int poll_ret = px4_poll(fds, 1, 1000);

    /* handle the poll result */
    if (poll_ret == 0) {
        /* this means none of our providers is giving us data */
        PX4_ERR("Got no data within a second");

    } else if (poll_ret < 0) {
        /* this is seriously bad - should be an emergency */
        if (error_counter < 10 || error_counter % 50 == 0) {
            /* use a counter to prevent flooding (and slowing us down) */
            PX4_ERR("ERROR return value from poll(): %d", poll_ret);
        }
    }

    error_counter++;

} else {

    if (fds[0].revents & POLLIN) {
        /* obtained data for the first file descriptor */
        struct sensor_combined_s raw;
        /* copy sensors raw data into local buffer */
        orb_copy(ORB_ID(sensor_combined), sensor_sub_fd, &raw);
        PX4_INFO("Accelerometer:\t%8.4f\t%8.4f\t%8.4f",
                 (double)raw.accelerometer_m_s2[0],
                 (double)raw.accelerometer_m_s2[1],
                 (double)raw.accelerometer_m_s2[2]);

        /* set att and publish this information for other apps */
        att.roll = raw.accelerometer_m_s2[0];
        att.pitch = raw.accelerometer_m_s2[1];
        att.yaw = raw.accelerometer_m_s2[2];
        orb_publish(ORB_ID(vehicle_attitude), att_pub, &att);
    }
}

```

```
    /* there could be more file descriptors here, in the form like:  
     * if (fds[1..n].revents & POLLIN) {}  
     */  
}  
}  
  
PX4_INFO("exiting");  
  
return 0;  
}
```

Running the final example

And finally run your app:

```
px4_simple_app
```

If you start QGroundControl, you can check the sensor values in the realtime plot (Tools -> Analyze)

Wrap-Up

This tutorial covered everything needed to develop a "grown up" PX4 autopilot application. Keep in mind that the full list of uORB messages / topics is [available here](#) and that the headers are well documented and serve as reference.

Video streaming in QGroundControl

This page shows how to set up a companion computer (Odroid C1) with a camera (Logitech C920) such that the video stream is transferred via the Odroid C1 to a network computer and displayed in the application QGroundControl that runs on this computer.

The whole hardware setup is shown in the figure below. It consists of the following parts:

- Odroid C1
- Logitech camera C920
- WiFi module TP-LINK TL-WN722N

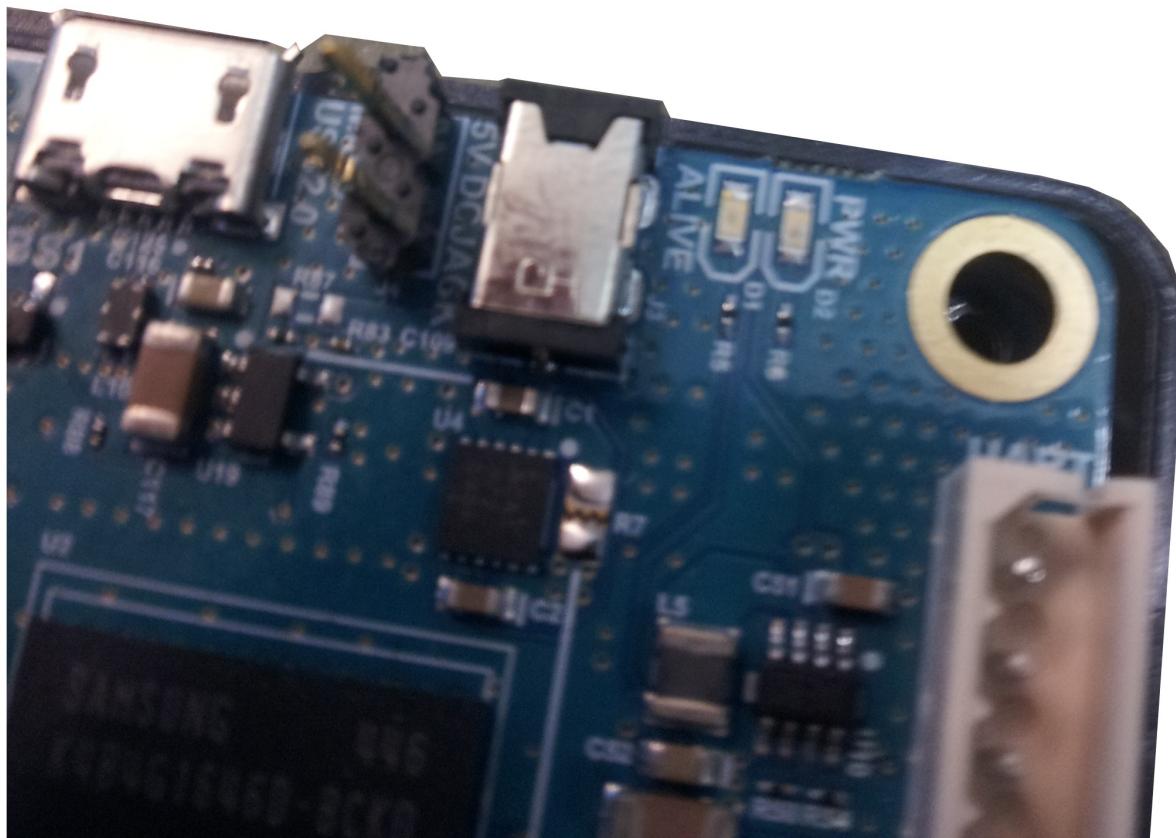


Install Linux environment in Odroid C1

To install the Linux environment (Ubuntu 14.04), follow the instruction given in the [Odroid C1 tutorial](#). In this tutorial it is also shown how to access the Odroid C1 with a UART cable and how to establish Ethernet connection.

Set up alternative power connection

The Odroid C1 can be powered via the 5V DC jack. If the Odroid is mounted on a drone, it is recommended to solder two pins next to the 5V DC jack by applying the through-hole soldering [method](#) as shown in the figure below. The power is delivered by connecting the DC voltage source (5 V) via a jumper cable (red in the image above) with the Odroid C1 and connect the ground of the circuit with a jumper cable (black in the image above) with a ground pin of the Odroid C1 in the example setup.



Enable WiFi connection for Odroid C1

In this this tutorial the WiFi module TP-LINK TL-WN722N is used. To enable WiFi connection for the Odroid C1, follow the steps described in the [Odroid C1 tutorial](#) in the section Establishing wifi connection with antenna.

Configure as WiFi Access Point

This sections shows how to set up the Odroid C1 such that it is an access point. The content is taken from this [tutorial](#) with some small adaptions. To enable to stream the video from the camera via the Odroid C1 to the QGroundControl that runs on a computer it is not required to follow this section. However, it is shown here because setting up the Odroid C1 as an access point allows to use the system in a stand-alone fashion. The TP-LINK TL-WN722N is used as a WiFi module. In the ensuing steps it is assumed that the Odroid C1 assigns the name wlan0 to your WiFi module. Change all occurrences of wlan0 to the appropriate interface if different (e.g. wlan1).

Onboard Computer as Access Point

For a more in depth explanation, you can look at [RPI-Wireless-Hotspot](#)

Install the necessary software

```
sudo apt-get install hostapd udhcpd
```

Configure DHCP. Edit the file `/etc/udhcpd.conf`

```
start 192.168.2.100 # This is the range of IPs that the hotspot will give to client devices.
end 192.168.2.200
interface wlan0 # The device uDHCP listens on.
remaining yes
opt dns 8.8.8.8 4.2.2.2 # The DNS servers client devices will use (if routing through the ethernet link).
opt subnet 255.255.255.0
opt router 192.168.2.1 # The Onboard Computer's IP address on wlan0 which we will set up shortly.
opt lease 864000 # 10 day DHCP lease time in seconds
```

All other 'opt' entries should be disabled or configured properly if you know what you are doing.

Edit the file `/etc/default/udhcpd` and change the line:

```
DHCPD_ENABLED="no"
```

to

```
#DHCPD_ENABLED="no"
```

You will need to give the Onboard Computer a static IP address. Edit the file

/etc/network/interfaces and replace the line iface wlan0 inet dhcp (or iface wlan0 inet manual) to:

```
auto wlan0
iface wlan0 inet static
address 192.168.2.1
netmask 255.255.255.0
network 192.168.2.0
broadcast 192.168.2.255
wireless-power off
```

Disable the original (WiFi Client) auto configuration. Change the lines (they probably will not be all next to each other or may not even be there at all):

```
allow-hotplug wlan0
wpa-roam /etc/wpa_supplicant/wpa_supplicant.conf
iface default inet dhcp
```

to:

```
#allow-hotplug wlan0
#wpa-roam /etc/wpa_supplicant/wpa_supplicant.conf
#iface default inet dhcp
```

If you have followed the [Odroid C1 tutorial](#) to set up the WiFi connection, you might have created the file /etc/network/intefaces.d/wlan0 . Please comment out all lines in that file such that those configurations have no effect anymore.

Configure HostAPD: To create a WPA-secured network, edit the file

/etc/hostapd/hostapd.conf (create it if it does not exist) and add the following lines:

```

auth_algs=1
channel=6           # Channel to use
hw_mode=g
ieee80211n=1       # 802.11n assuming your device supports it
ignore_broadcast_ssid=0
interface=wlan0
wpa=2
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
# Change the to the proper driver
driver=nl80211
# Change these to something else if you want
ssid=OdroidC1
wpa_passphrase=QGroundControl

```

Change `ssid=`, `channel=`, and `wpa_passphrase=` to values of your choice. SSID is the hotspot's name which is broadcast to other devices, channel is what frequency the hotspot will run on, `wpa_passphrase` is the password for the wireless network. For many more options see the file `/usr/share/doc/hostapd/examples/hostapd.conf.gz`. Look for a channel that is not in use in the area. You can use tools such as wavemon for that.

Edit the file `/etc/default/hostapd` and change the line:

```
#DAEMON_CONF=""
```

to:

```
DAEMON_CONF="/etc/hostapd/hostapd.conf"
```

Your Onboard Computer should now be hosting a wireless hotspot. To get the hotspot to start on boot, run these additional commands:

```

sudo update-rc.d hostapd enable
sudo update-rc.d udhcpcd enable

```

This is enough to have the Onboard Computer present itself as an Access Point and allow your ground station to connect. If you truly want to make it work as a real Access Point (routing the WiFi traffic to the Onboard Computer's ethernet connection), we need to configure the routing and network address translation (NAT). Enable IP forwarding in the kernel:

```
sudo sh -c "echo 1 > /proc/sys/net/ipv4/ip_forward"
```

To enable NAT in the kernel, run the following commands:

```
sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
sudo iptables -A FORWARD -i eth0 -o wlan0 -m state --state RELATED,ESTABLISHED -j ACCEPT
sudo iptables -A FORWARD -i wlan0 -o eth0 -j ACCEPT
```

To make this permanent, run the following command:

```
sudo sh -c "iptables-save > /etc/iptables.ipv4.nat"
```

Now edit the file `/etc/network/interfaces` and add the following line to the bottom of the file:

```
up iptables-restore < /etc/iptables.ipv4.nat
```

gstreamer Installation

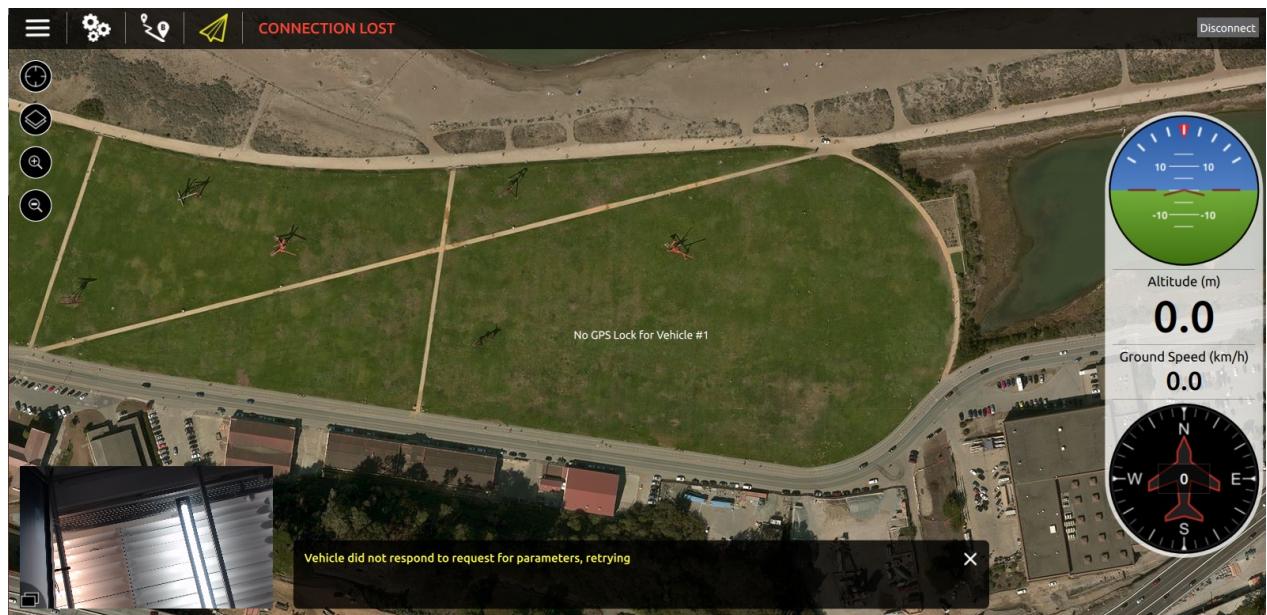
To install gstreamer packages on the computer and on the Odroid C1 and start the stream, follow the instruction given in the [QGroundControl README](#).

If you cannot start the stream on the Odroid with the `uvch264s` plugin, you can also try to start it with the `v4l2src` plugin:

```
gst-launch-1.0 v4l2src device=/dev/video0 ! video/x-h264,width=1920,height=1080,framerate=24/1 ! h264parse ! rtph264pay ! udpsink host=xxx.xxx.xxx.xxx port=5000
```

Where `xxx.xxx.xxx.xxx` is the IP address where QGC is running. If you get the system error: `Permission denied`, you might need to prepend `sudo` to the command above.

If everything works, you should see the video stream on the bottom left corner in the flight-mode window of QGroundControl as shown in the screenshot below.



If you click on the video stream, the satellite map is shown in the left bottom corner and the video is shown in the whole background.

Optical Flow and LIDAR

This page shows you how to set up the PX4Flow and a LIDAR distance measurement device for position estimation. Using a LIDAR device is not necessary if you use the LPE estimator described below since the PX4FLOW has a sonar, but LIDAR does improve performance.

Selecting an Estimator

Two estimators support optical flow based navigation, LPE and INAV. There are benefits to both, but LPE is currently recommended for new users as it has the most testing and is the most robust. INAV uses slightly less CPU.

Use the `SYS_MC_EST_GROUP` parameter to select the estimator and then reboot.

Hardware

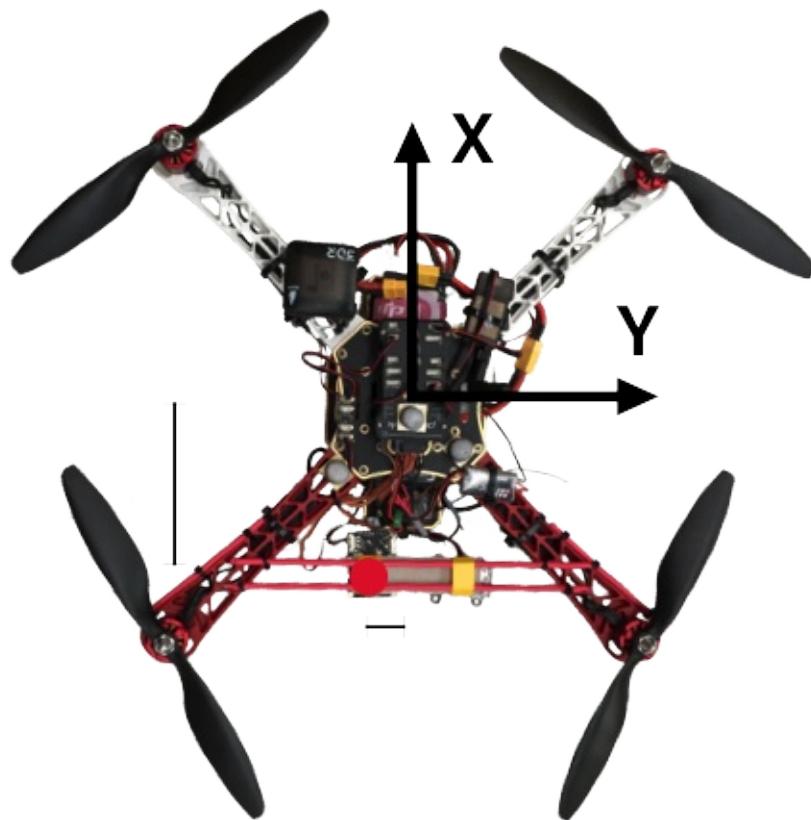


Figure 1: Mounting Coordinate Frame (relevant to parameters below)

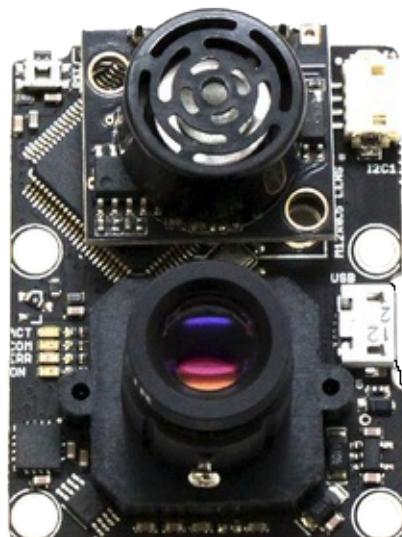


Figure 2: PX4Flow optical flow sensor (camera and sonar)

The PX4Flow has to point towards the ground and can be connected using the I2C port on the pixhawk. For best performance make sure the PX4Flow is attached at a good position and is not exposed to vibration. (preferably on the down side of the quad-rotor).

Note: The default orientation is that the PX4Flow sonar side (+Y on flow) be pointed toward +X on the vehicle (forward). If it is not, you will need to set SENS_FLOW_ROT accordingly.



Figure 3: Lidar Lite

Several LIDAR options exist including the Lidar-Lite (not currently manufactured) and the sf10a: [sf10a](#). For the connection of the LIDAR-Lite please refer to [this](#) page. The sf10a can be connected using a serial cable.

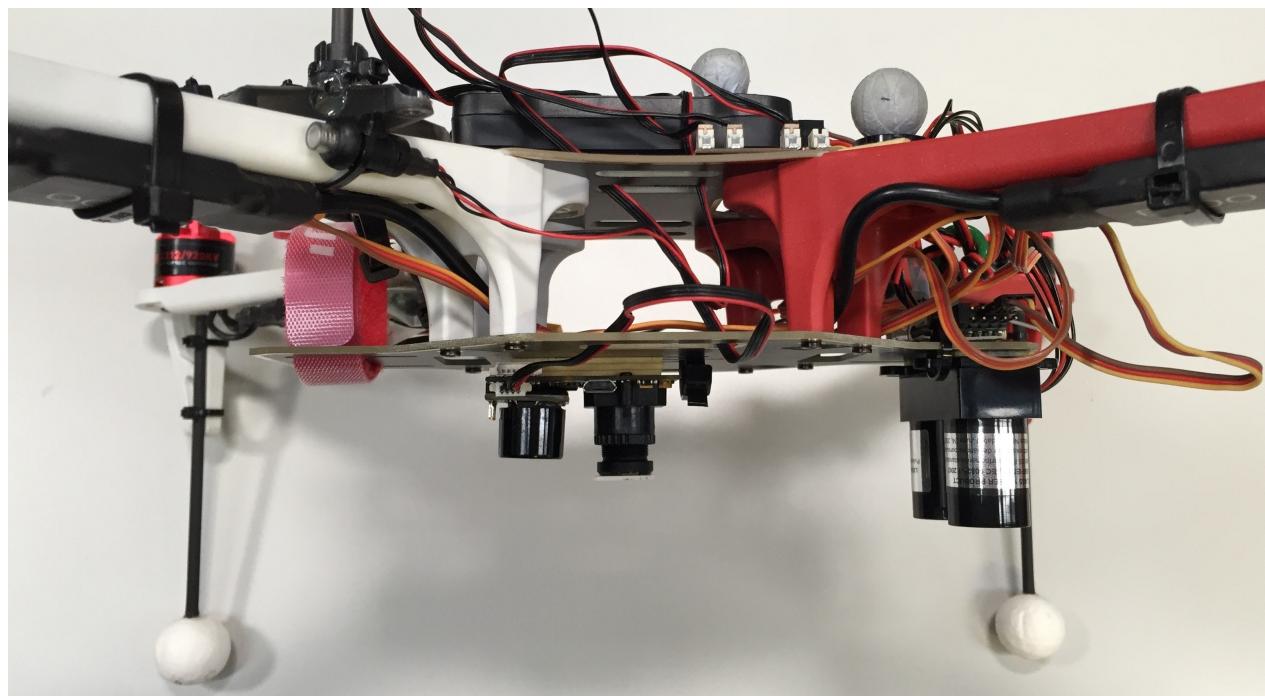


Figure: PX4Flow/ Lidar-Lite mounting DJI F450



Figure: This Iris+ has a PX4Flow attached without a LIDAR, this works with the LPE estimator.



Figure: A weather-proof case was constructed for this flow unit. Foam is also used to surround the sonar to reduce prop noise read by the sonar and help protect the camera lens from crashes.

Focusing Camera

In order to ensure good optical flow quality, it is important to focus the camera on the PX4Flow to the desired height of flight. To focus the camera, put an object with text on (e. g. a book) and plug in the PX4Flow into usb and run QGroundControl. Under the settings menu, select the PX4Flow and you should see a camera image. Focus the lens by unscrewing the set screw and loosening and tightening the lens to find where it is in focus.

Note: If you fly above 3m, the camera will be focused at infinity and won't need to be changed for higher flight.



Figure: Use a text book to focus the flow camera at the height you want to fly, typically 1-3 meters. Above 3 meters the camera should be focused at infinity and work for all higher altitudes.

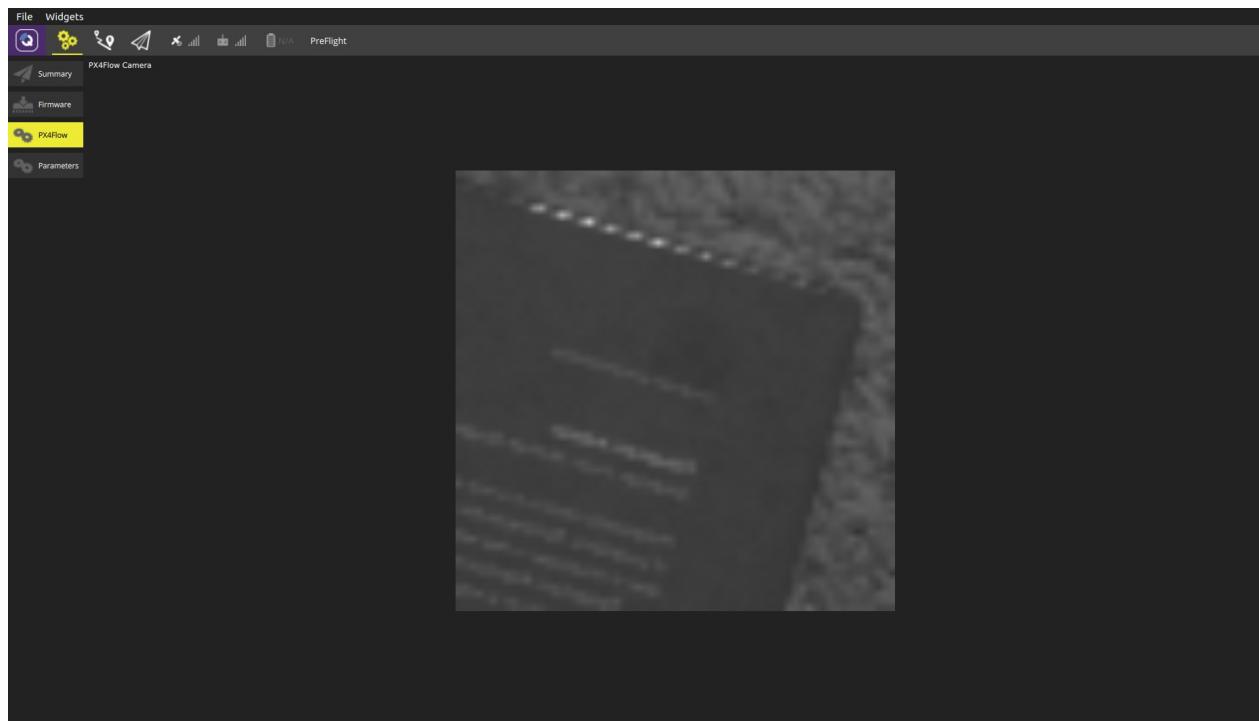


Figure: The px4flow interface in QGroundControl that can be used for focusing the camera

Sensor Parameters

All the parameters can be changed in QGroundControl

- SENS_EN_LL40LS Set to 1 to enable lidar-lite distance measurements
- SENS_EN_SF0X Set to 1 to enable lightware distance measurements (e.g. sf02 and sf10a)

Local Position Estimator (LPE)

LPE is an Extended Kalman Filter based estimator for position and velocity states. It uses inertial navigation and is similar to the INAV estimator below but it dynamically calculates the Kalman gain based on the state covariance. It also is capable of detecting faults, which is beneficial for sensors like sonar which can return invalid reads over soft surfaces.

Flight Video Indoor



[Video link](#)

Flight Video Outdoor



[Video link](#)

For outdoor autonomous missions with LPE estimator, see tutorial on (Optical Flow Outdoors)
[./optical-flow-outdoors.md]

Parameters

The local position estimator will automatically fuse LIDAR and optical flow data when the sensors are plugged in.

- LPE_FLOW_OFF_Z - This is the offset of the optical flow camera from the center of mass of the vehicle. This measures positive down and defaults to zero. This can be left zero for most typical configurations where the z offset is negligible.
- LPE_FLW_XY - Flow standard deviation in meters.
- LPW_FLW_QMIN - Minimum flow quality to accept measurement.
- LPE_SNR_Z - Sonar standard deviation in meters.
- LPE_SNR_OFF_Z - Offset of sonar sensor from center of mass.
- LPE_LDR_Z - Lidar standard deviation in meters.
- LPE_LDR_Z_OFFSET - Offset of lidar from center of mass.
- LPE_GPS_ON - You won't be able to fly without GPS if LPE_GPS_ON is set to 1. You must disable it or it will wait for GPS altitude to initialize position. This is so that GPS altitude will take precedence over baro altitude if GPS is available.

NOTE: LPE_GPS_ON must be set to 0 to enable flight without GPS

Inertial Navigation Estimator (INAV)

INAV has a fixed gain matrix for correction and can be viewed as a steady state Kalman filter. It has the lowest computational cost of all position estimators.

Flight Video Indoor



[Video link](#)

Flight Video Outdoor



[Video link](#)

Parameters

- INAV_LIDAR_EST Set to 1 to enable altitude estimation based on distance measurements
- INAV_FLOW_DIST_X and INAV_FLOW_DIST_Y These two values (in meters) are used for yaw compensation. The offset has to be measured according to Figure 1 above. In the above example the offset of the PX4Flow (red dot) would have a negative X offset and a negative Y offset.
- INAV_LIDAR_OFF Set a calibration offset for the lidar-lite in meters. The value will be added to the measured distance.

Advanced Parameters

For advanced usage/development the following parameters can be changed as well. Do NOT change them if you do not know what you are doing!

- INAV_FLOW_W Sets the weight for the flow estimation/update
- INAV_LIDAR_ERR Sets the threshold for altitude estimation/update in meters. If the correction term is bigger than this value, it will not be used for the update.

Integration Testing

This is about end to end integration testing. Tests are executed automatically ([Jenkins CI](#))

ROS / MAVROS Tests

Prerequisites:

- [SITL Simulation](#)
- [Gazebo](#)
- [ROS and MAVROS](#)

Execute Tests

To run the complete MAVROS test suite:

```
cd <Firmware_clone>
source integrationtests/setup_gazebo_ros.bash $(pwd)
rostest px4 mavros_posix_tests_iris.launch
```

Or with GUI to see what's happening:

```
rostest px4 mavros_posix_tests_iris.launch gui:=true headless:=false
```

Write a new MAVROS test (Python)

Currently in early stages, more streamlined support for testing (helper classes/methods etc.) to come.

1.) Create a new test script

Test scripts are located in `integrationtests/python_src/px4_it/mavros/`. See other existing scripts for examples. Also please consult the official ROS documentation on how to use [unittest](#).

Empty test skeleton:

```

#!/usr/bin/env python
# [... LICENSE ...]

#
# @author Example Author <author@example.com>
#
PKG = 'px4'

import unittest
import rospy
import rosbag

from sensor_msgs.msg import NavSatFix

class MavrosNewTest(unittest.TestCase):
    """
    Test description
    """

    def setUp(self):
        rospy.init_node('test_node', anonymous=True)
        rospy.wait_for_service('mavros/cmd/arm', 30)

        rospy.Subscriber("mavros/global_position/global", NavSatFix, self.global_position_callback)
        self.rate = rospy.Rate(10) # 10hz
        self.has_global_pos = False

    def tearDown(self):
        pass

    #
    # General callback functions used in tests
    #
    def global_position_callback(self, data):
        self.has_global_pos = True

    def test_method(self):
        """Test method description"""

        # FIXME: hack to wait for simulation to be ready
        while not self.has_global_pos:
            self.rate.sleep()

        # TODO: execute test

    if __name__ == '__main__':
        importrostest
        rostest.rosrun(PKG, 'mavros_new_test', MavrosNewTest)

```

2.) Run the new test only

```
# Start simulation
cd <Firmware_clone>
source integrationtests/setup_gazebo_ros.bash $(pwd)
roslaunch px4 mavros_posix_sitl.launch

# Run test (in a new shell):
cd <Firmware_clone>
source integrationtests/setup_gazebo_ros.bash $(pwd)
rosrun px4 mavros_new_test.py
```

3.) Add new test node to launch file

In `launch/mavros_posix_tests_irisl.launch` add new entry in test group:

```
<group ns="$(arg ns)">
  [...]
  <test test-name="mavros_new_test" pkg="px4" type="mavros_new_test.py" />
</group>
```

Run the complete test suite as described above.

Optical Flow Outdoors

This page shows you how to set up the PX4Flow for position estimation and autonomous flight outdoors. Using a LIDAR device is not necessary, but LIDAR does improve performance.

Selecting LPE Estimator

The only estimator that is tested to work with optical flow based autonomous flight outdoors is, LPE.

Use the `SYS_MC_EST_GROUP = 1` parameter to select the estimator and then reboot.

Hardware

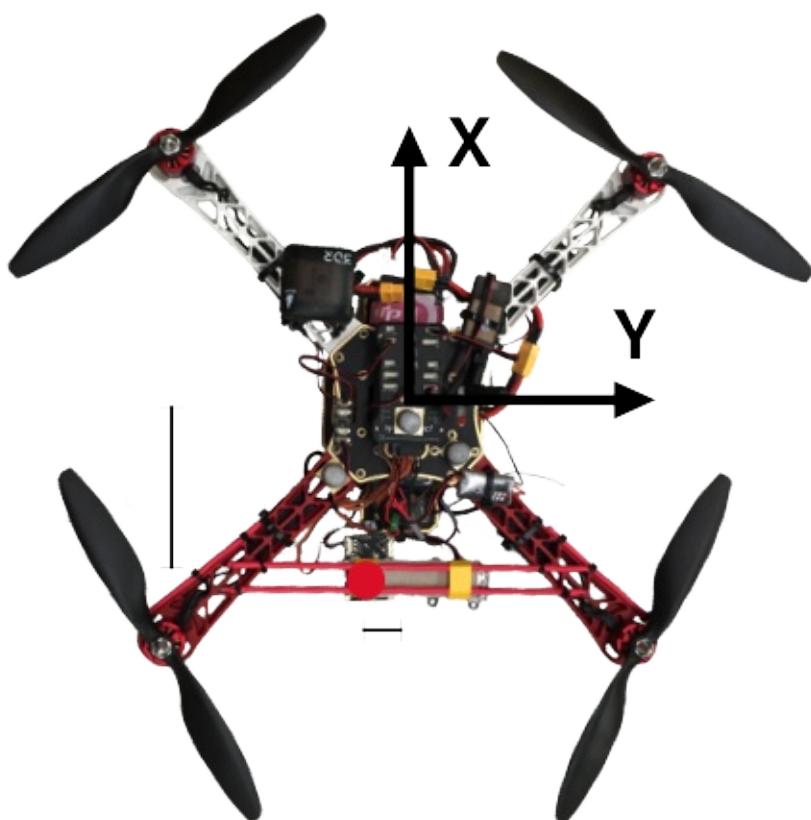


Figure 1: Mounting Coordinate Frame (relevant to parameters below)

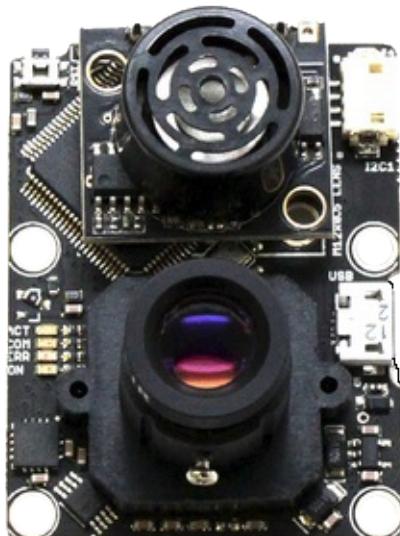


Figure 2: PX4Flow optical flow sensor (camera and sonar)

The PX4Flow has to point towards the ground and can be connected using the I2C port on the pixhawk. For best performance make sure the PX4Flow is attached at a good position and is not exposed to vibration. (preferably on the down side of the quad-rotor).

Note: The default orientation is that the PX4Flow sonar side (+Y on flow) be pointed toward +X on the vehicle (forward). If it is not, you will need to set SENS_FLOW_ROT accordingly.



Figure 3: Lidar Lite

Several LIDAR options exist including the Lidar-Lite (not currently manufactured) and the sf10a: [sf10a](#). For the connection of the LIDAR-Lite please refer to [this](#) page. The sf10a can be connected using a serial cable.

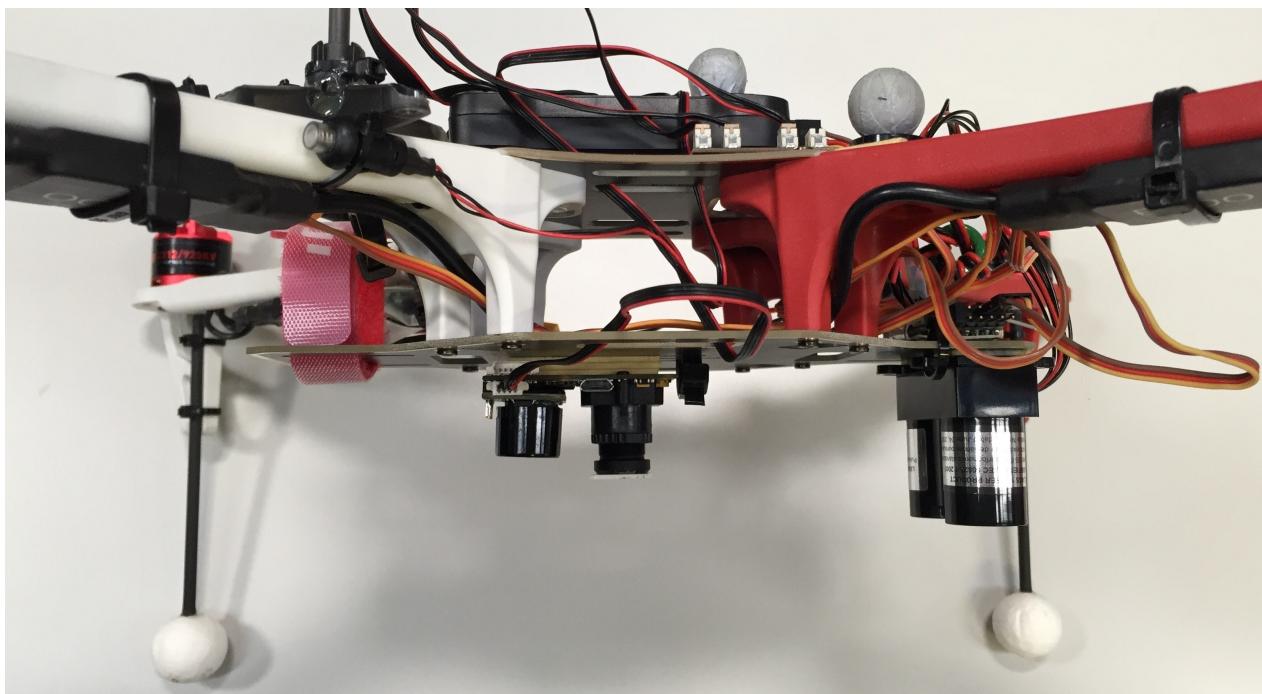


Figure: PX4Flow/ Lidar-Lite mounting DJI F450



Figure: This Iris+ has a PX4Flow attached without a LIDAR, this works with the LPE estimator.



Figure: A weather-proof case was constructed for this flow unit. Foam is also used to surround the sonar to reduce prop noise read by the sonar and help protect the camera lens from crashes.

Focusing Camera

In order to ensure good optical flow quality, it is important to focus the camera on the PX4Flow to the desired height of flight. To focus the camera, put an object with text on (e. g. a book) and plug in the PX4Flow into usb and run QGroundControl. Under the settings menu, select the PX4Flow and you should see a camera image. Focus the lens by unscrewing the set screw and loosening and tightening the lens to find where it is in focus.

Note: If you fly above 3m, the camera will be focused at infinity and won't need to be changed for higher flight.



Figure: Use a text book to focus the flow camera at the height you want to fly, typically 1-3 meters. Above 3 meters the camera should be focused at infinity and work for all higher altitudes.

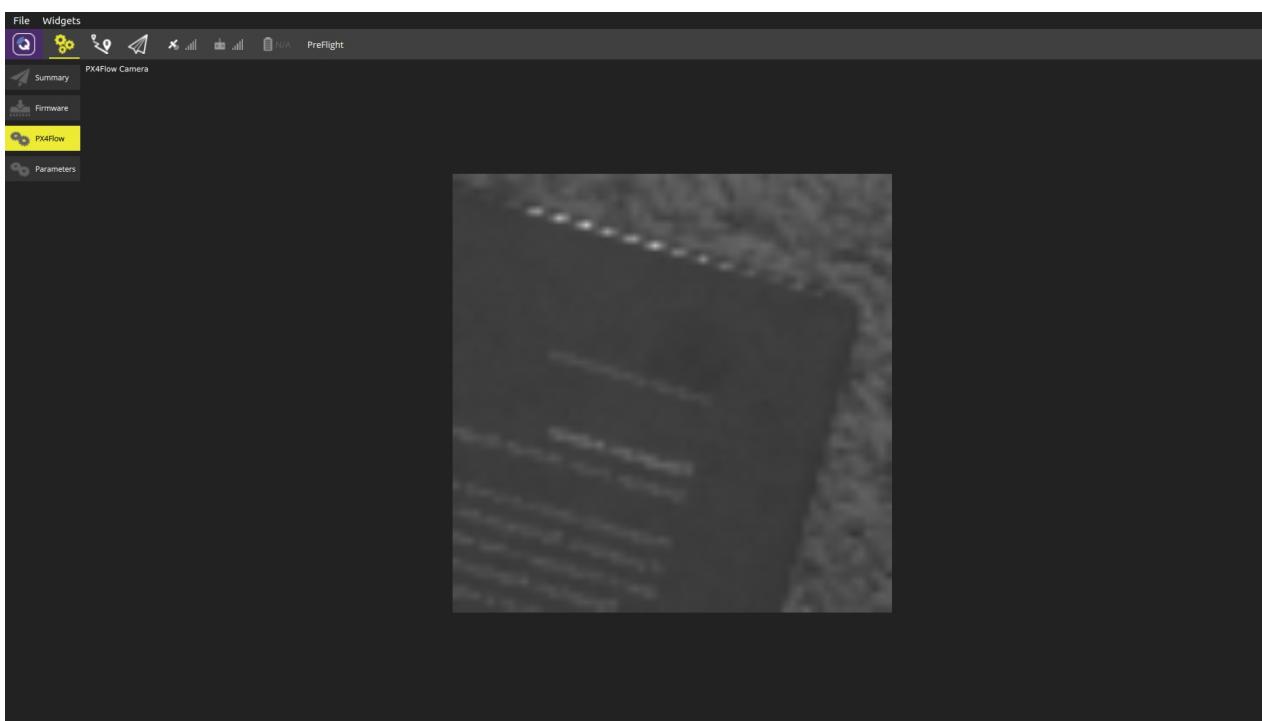


Figure: The px4flow interface in QGroundControl that can be used for focusing the camera

Sensor Parameters

All the parameters can be changed in QGroundControl

- SENS_EN_LL40LS Set to 1 to enable lidar-lite distance measurements

- SENS_EN_SF0X Set to 1 to enable lightware distance measurements (e.g. sf02 and sf10a)

Local Position Estimator (LPE)

LPE is an Extended Kalman Filter based estimator for position and velocity states. It uses inertial navigation and is similar to the INAV estimator below but it dynamically calculates the Kalman gain based on the state covariance. It also is capable of detecting faults, which is beneficial for sensors like sonar which can return invalid reads over soft surfaces.

Flight Video Outdoor



[Video link](#)

Below is a plot of the autonomous mission from the outdoor flight video above using optical flow. GPS is not used to estimate the vehicle position but is plotted for a ground truth comparison. The offset between the GPS and flow data is due to the initialization of the estimator from user error on where it was placed. The initial placement is assumed to be at LPE_LAT and LPE_LON (described below).

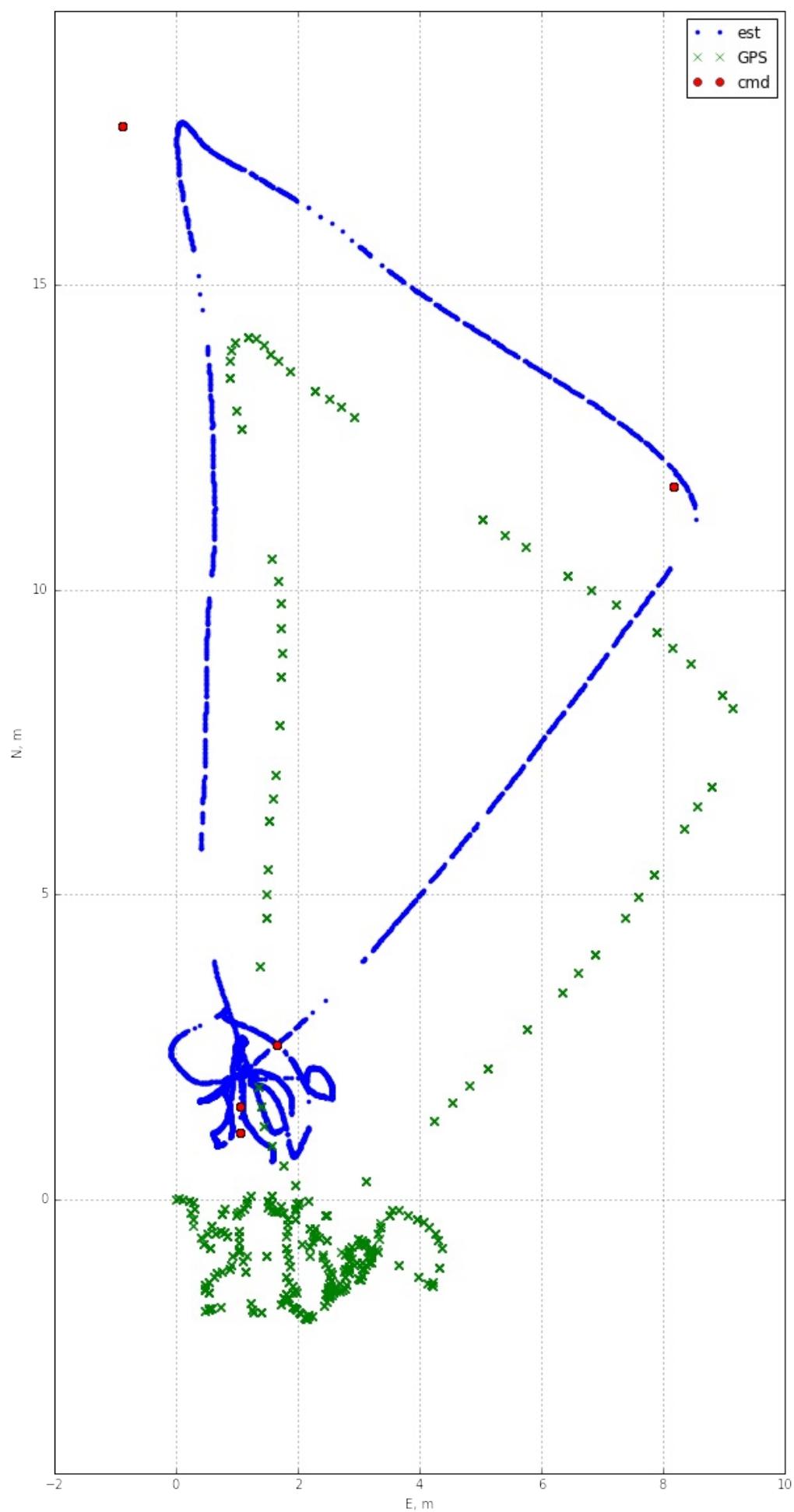


Figure 4: LPE based autonomous mission with optical flow and sonar

Parameters

The local position estimator will automatically fuse LIDAR and optical flow data when the sensors are plugged in.

- LPE_FLOW_OFF_Z - This is the offset of the optical flow camera from the center of mass of the vehicle. This measures positive down and defaults to zero. This can be left zero for most typical configurations where the z offset is negligible.
- LPE_FLW_XY - Flow standard deviation in meters.
- LPW_FLW_QMIN - Minimum flow quality to accept measurement.
- LPE_SNR_Z - Sonar standard deviation in meters.
- LPE_SNR_OFF_Z - Offset of sonar sensor from center of mass.
- LPE_LDR_Z - Lidar standard deviation in meters.
- LPE_LDR_Z_OFFSET - Offset of lidar from center of mass.
- LPE_GPS_ON - You won't be able to fly without GPS if LPE_GPS_ON is set to 1. You must disable it or it will wait for GPS altitude to initialize position. This is so that GPS altitude will take precedence over baro altitude if GPS is available.

NOTE: LPE_GPS_ON must be set to 0 to enable flight without GPS

Autonomous Flight Parameters

Tell the vehicle where it is in the world

- LPE_LAT - The latitude associated with the (0,0) coordinate in the local frame.
- LPE_LON - The longitude associated with the (0,0) coordinate in the local frame.

Make the vehicle keep a low altitude and slow speed

- MPC_ALT_MODE - Set this to 1 to enable terrain follow
- LPE_T_Z - This is the terrain process noise. If your environment is hilly, set it to 0.1, if it is a flat parking lot etc. set it to 0.01.
- MPC_XY_VEL_MAX - Set this to 2 to limit leaning
- MPC_XY_P - Decrease this to around 0.5 to limit leaning
- MIS_TAKEOFF_ALT - Set this to 2 meters to allow low altitude takeoffs.

Waypoints

- Create waypoints with altitude 3 meters or below.
- Do not create flight plans with extremely long distance, expect about 1m drift / 100 m of flight.

Note: Before your first auto flight, walk the vehicle manually through the flight with the flow sensor to make sure it will trace the path you expect.

Using the ecl EKF

This tutorial answers common questions about use of the ECL EKF algorithm.

What is the ecl EKF?

The ECL (Estimation and Control Library) uses an Extended Kalman Filter algorithm to process sensor measurements and provide an estimate of the following states:

- Quaternion defining the rotation from North,East,Down local earth fram to X,Y,Z body frame
- Velocity at the IMU - North,East,Down (m/s)
- Position at the IMU - North,East,Down (m)
- IMU delta angle bias estimates - X,Y,Z (rad)
- IMU delta velocity bias estimates - X,Y,Z(m/s)
- Earth Magnetic field components - North,East,Down (gauss)
- Vehicle body frame magnetic field bias - X,Y,Z (gauss)
- Wind velocity - North,East (m/s)

The EKF runs on a delayed 'fusion time horizon' to allow for different time delays on each measurement relative to the IMU. Data for each sensor is FIFO buffered and retrieved from the buffer by the EKF to be used at the correct time. The delay compensation for each sensor is controlled by the EKF2_*_DELAY parameters.

A complementary filter is used to propagate the states forward from the 'fusion time horizon' to current time using the buffered IMU data. The time constant for this filter is controlled by the EKF2_TAU_VEL and EKF2_TAU_POS parameters.

Note: The 'fusion time horizon' delay and length of the buffers is determined by the largest of the EKF2_*_DELAY parameters. If a sensor is not being used, it is recommended to set its time delay to zero. Reducing the 'fusion time horizon' delay reduces errors in the complementary filter used to propagate states forward to current time.

The position and velocity states are adjusted to account for the offset between the IMU and the body frame before they are output to the control loops. The position of the IMU relative to the body frame is set by the EKF2_IMU_POS_X,Y,Z parameters.

The EKF uses the IMU data for state prediction only. IMU data is not used as an observation in the EKF derivation. The algebraic equations for the covariance prediction, state update and covariance update were derived using the Matlab symbolic toolbox and can be found

here: [Matlab Symbolic Derivation](#)

What sensor measurements does it use?

The EKF has different modes of operation that allow for different combinations of sensor measurements. On start-up the filter checks for a minimum viable combination of sensors and after initial tilt, yaw and height alignment is completed, enters a mode that provides rotation, vertical velocity, vertical position, IMU delta angle bias and IMU delta velocity bias estimates.

This mode requires IMU data, a source of yaw (magnetometer or external vision) and a source of height data. This minimum data set is required for all EKF modes of operation. Other sensor data can then be used to estimate additional states.

IMU

- Three axis body fixed Inertial Measurement unit delta angle and delta velocity data at a minimum rate of 100Hz. Note: Coning corrections should be applied to the IMU delta angle data before it is used by the EKF.

Magnetometer

Three axis body fixed magnetometer data (or external vision system pose data) at a minimum rate of 5Hz is required. Magnetometer data can be used in two ways:

- Magnetometer measurements are converted to a yaw angle using the tilt estimate and magnetic declination. This yaw angle is then used as an observation by the EKF. This method is less accurate and does not allow for learning of body frame field offsets, however it is more robust to magnetic anomalies and large start-up gyro biases. It is the default method used during start-up and on ground.
- The XYZ magnetometer readings are used as separate observations. This method is more accurate and allows body frame offsets to be learned, but assumes the earth magnetic field environment only changes slowly and performs less well when there are significant external magnetic anomalies. It is the default method when the vehicle is airborne and has climbed past 1.5 m altitude.

The logic used to select the mode is set by the EKF2_MAG_TYPE parameter.

Height

A source of height data - either GPS, barometric pressure, range finder or external vision at a minimum rate of 5Hz is required. Note: The primary source of height data is controlled by the EKF2_HGT_MODE parameter.

If these measurements are not present, the EKF will not start. When these measurements have been detected, the EKF will initialise the states and complete the tilt and yaw alignment. When tilt and yaw alignment is complete, the EKF can then transition to other modes of operation enabling use of additional sensor data:

GPS

GPS measurements will be used for position and velocity if the following conditions are met:

- GPS use is enabled via setting of the EKF2_AID_MASK parameter.
- GPS quality checks have passed. These checks are controlled by the EKF2_GPS_CHECK and EKF2_REQ<> parameters.
- GPS height can be used directly by the EKF via setting of the EKF2_HGT_MODE parameter.

Range Finder

Range finder distance to ground is used a by a single state filter to estimate the vertical position of the terrain relative to the height datum.

If operating over a flat surface that can be used as a zero height datum, the range finder data can also be used directly by the EKF to estimate height by setting the EKF2_HGT_MODE parameter to 2.

Airspeed

Equivalent Airspeed (EAS) data can be used to estimate wind velocity and reduce drift when GPS is lost by setting EKF2_ARSP_THR to a positive value. Airspeed data will be used when it exceeds the threshold set by a positive value for EKF2_ARSP_THR and the vehicle type is not rotary wing.

Synthetic Sideslip

Fixed wing platforms can take advantage of an assumed sideslip observation of zero to improve wind speed estimation and also enable wind speed estimation without an airspeed sensor. This is enabled by setting the EKF2_FUSE_BETA parameter to 1.

Optical Flow

Optical flow data will be used if the following conditions are met:

- Valid range finder data is available.
- Bit position 1 in the EKF2_AID_MASK parameter is true.
- The quality metric returned by the flow sensor is greater than the minimum requirement set by the EKF2_OF_QMIN parameter

External Vision System

Position and Pose Measurements from an external vision system, eg Vicon, can be used:

- External vision system horizontal position data will be used if bit position 3 in the EKF2_AID_MASK parameter is true.
- External vision system vertical position data will be used if the EKF2_HGT_MODE parameter is set to 3.
- External vision system pose data will be used for yaw estimation if bit position 4 in the EKF2_AID_MASK parameter is true.

How do I use the 'ecl' library EKF?

Set the SYS_MC_EST_GROUP parameter to 2 to use the ecl EKF.

What are the advantages and disadvantages of the ecl EKF over other estimators?

Like all estimators, much of the performance comes from the tuning to match sensor characteristics. Tuning is a compromise between accuracy and robustness and although we have attempted to provide a tune that meets the needs of most users, there will be applications where tuning changes are required.

For this reason, no claims for accuracy relative to the legacy combination of attitude_estimator_q + local_position_estimator have been made and the best choice of estimator will depend on the application and tuning.

Disadvantages

- The ecl EKF is a complex algorithm that requires a good understanding of extended Kalman filter theory and its application to navigation problems to tune successfully. It is therefore more difficult for users that are not achieving good results to know what to change.

- The ecl EKF uses more RAM and flash space
- The ecl EKF uses more logging space.
- The ecl EKF has had less flight time

Advantages

- The ecl EKF is able to fuse data from sensors with different time delays and data rates in a mathematically consistent way which improves accuracy during dynamic manoeuvres once time delay parameters are set correctly.
- The ecl EKF is capable of fusing a large range of different sensor types.
- The ecl EKF detects and reports statistically significant inconsistencies in sensor data, assisting with diagnosis of sensor errors.
- For fixed wing operation, the ecl EKF estimates wind speed with or without an airspeed sensor and is able to use the estimated wind in combination with airspeed measurements and sideslip assumptions to extend the dead-reckoning time available if GPS is lost in flight.
- The ecl EKF estimates 3-axis accelerometer bias which improves accuracy for tailsitters and other vehicles that experience large attitude changes between flight phases.
- The federated architecture (combined attitude and position/velocity estimation) means that attitude estimation benefits from all sensor measurements. This should provide the potential for improved attitude estimation if tuned correctly.

How do I check the EKF performance?

EKF outputs, states and status data are published to a number of uORB topics which are logged to the SD card during flight. The following guide assumes that data has been logged using the .ulog file format. To use the .ulog format, set the SYS_LOGGER parameter to 1.

The .ulog format data can be parsed in python by using the [PX4 pyulog library](#).

Most of the EKF data is found in the [ekf2_innovations](#) and [estimator_status](#) uORB messages that are logged to the .ulog file.

Output Data

- Attitude output data is found in the [vehicle_attitude](#) message.
- Local position output data is found in the [vehicle_local_position](#) message.
- Control loop feedback data is found in the [control_state](#) message.
- Global (WGS-84) output data is found in the [vehicle_global_position](#) message.
- Wind velocity output data is found in the [wind_estimate](#) message.

States

Refer to states[32] in [estimator_status](#). The index map for states[32] is as follows:

- [0 ... 3] Quaternions
- [4 ... 6] Velocity NED (m/s)
- [7 ... 9] Position NED (m)
- [10 ... 12] IMU delta angle bias XYZ (rad)
- [13 ... 15] IMU delta velocity bias XYZ (m/s)
- [16 ... 18] Earth magnetic field NED (gauss)
- [19 ... 21] Body magnetic field XYZ (gauss)
- [22 ... 23] Wind velocity NE (m/s)
- [24 ... 32] Not Used

State Variances

Refer to covariances[28] in [estimator_status](#). The index map for covariances[28] is as follows:

- [0 ... 3] Quaternions
- [4 ... 6] Velocity NED (m/s)²
- [7 ... 9] Position NED (m²)
- [10 ... 12] IMU delta angle bias XYZ (rad²)
- [13 ... 15] IMU delta velocity bias XYZ (m/s)²
- [16 ... 18] Earth magnetic field NED (gauss²)
- [19 ... 21] Body magnetic field XYZ (gauss²)
- [22 ... 23] Wind velocity NE (m/s)²
- [24 ... 28] Not Used

Observation Innovations

- Magnetometer XYZ (gauss) : Refer to mag_innov[3] in [ekf2_innovations](#).
- Yaw angle (rad) : Refer to heading_innov in [ekf2_innovations](#).
- Velocity and position innovations : Refer to vel_pos_innov[6] in [ekf2_innovations](#). The index map for vel_pos_innov[6] is as follows:
 - [0 ... 2] Velocity NED (m/s)
 - [3 ... 5] Position NED (m)
- True Airspeed (m/s) : Refer to airspeed_innov in [ekf2_innovations](#).
- Synthetic sideslip (rad) : Refer to beta_innov in [ekf2_innovations](#).
- Optical flow XY (rad/sec) : Refer to flow_innov in [ekf2_innovations](#).
- Height above ground (m) : Refer to hagl_innov in [ekf2_innovations](#).

Observation Innovation Variances

- Magnetometer XYZ (gauss²) : Refer to mag_innov_var[3] in [ekf2_innovations](#).
- Yaw angle (rad²) : Refer to heading_innov_var in the ekf2_innovations message.
- Velocity and position innovations : Refer to vel_pos_innov_var[6] in [ekf2_innovations](#).
The index map for vel_pos_innov_var[6] is as follows:
 - [0 ... 2] Velocity NED (m/s)²
 - [3 ... 5] Position NED (m²)
- True Airspeed (m/s)² : Refer to airspeed_innov_var in [ekf2_innovations](#).
- Synthetic sideslip (rad²) : Refer to beta_innov_var in [ekf2_innovations](#).
- Optical flow XY (rad/sec)² : Refer to flow_innov_var in [ekf2_innovations](#).
- Height above ground (m²) : Refer to hagl_innov_var in [ekf2_innovations](#).

Output Complementary Filter

The output complementary filter is used to propagate states forward from the fusion time horizon to current time. To check the magnitude of the angular, velocity and position tracking errors measured at the fusion time horizon, refer to output_tracking_error[3] in the ekf2_innovations message. The index map is as follows:

- [0] Angular tracking error magnitude (rad)
- [1] Velocity tracking error magnitude (m/s). The velocity tracking time constant can be adjusted using the EKF2_TAU_VEL parameter. Reducing this parameter reduces steady state errors but increases the amount of observation noise on the NED velocity outputs.
- [2] Position tracking error magnitude (m). The position tracking time constant can be adjusted using the EKF2_TAU_POS parameter. Reducing this parameter reduces steady state errors but increases the amount of observation noise on the NED position outputs.

EKF Errors

The EKF contains internal error checking for badly conditioned state and covariance updates. Refer to the filter_fault_flags in [estimator_status](#).

Observation Errors

There are two categories of observation faults:

- Loss of data. An example of this is a range finder failing to provide a return.
- The innovation, which is the difference between the state prediction and sensor observation is excessive. An example of this is excessive vibration causing a large

vertical position error, resulting in the barometer height measurement being rejected.

Both of these can result in observation data being rejected for long enough to cause the EKF to attempt a reset of the states using the sensor observations. All observations have a statistical confidence check applied to the innovations. The number of standard deviations for the check are controlled by the EKF2_<>_GATE parameter for each observation type.

Test levels are available in [estimator_status](#) as follows:

- mag_test_ratio : ratio of the largest magnetometer innovation component to the innovation test limit
- vel_test_ratio : ratio of the largest velocity innovation component to the innovation test limit
- pos_test_ratio : ratio of the largest horizontal position innovation component to the innovation test limit
- hgt_test_ratio : ratio of the vertical position innovation to the innovation test limit
- tas_test_ratio : ratio of the true airspeed innovation to the innovation test limit
- hagl_test_ratio : ratio of the height above ground innovation to the innovation test limit

For a binary pass/fail summary for each sensor, refer to innovation_check_flags in [estimator_status](#).

GPS Quality Checks

The EKF applies a number of GPS quality checks before commencing GPS aiding. These checks are controlled by the EKF2_GPS_CHECK and EKF2_REQ<> parameters. The pass/fail status for these checks is logged in the [estimator_status.gps_check_fail_flags](#) message. This integer will be zero when all required GPS checks have passed. If the EKF is not commencing GPS alignment, check the value of the integer against the bitmask definition [gps_check_fail_flags](#) in [estimator_status](#).

EKF Numerical Errors

The EKF uses single precision floating point operations for all of its computations and first order approximations for derivation of the covariance prediction and update equations in order to reduce processing requirements. This means that it is possible when re-tuning the EKF to encounter conditions where the covariance matrix operations become badly conditioned enough to cause divergence or significant errors in the state estimates.

To prevent this, every covariance and state update step contains the following error detection and correction steps:

- If the innovation variance is less than the observation variance (this requires a negative

state variance which is impossible) or the covariance update will produce a negative variance for any of the states, then:

- The state and covariance update is skipped
- The corresponding rows and columns in the covariance matrix are reset
- The failure is recorded in the `estimator_status.filter_fault_flags` message
- State variances (diagonals in the covariance matrix) are constrained to be non-negative.
- An upper limit is applied to state variances.
- Symmetry is forced on the covariance matrix.

After re-tuning the filter, particularly re-tuning that involve reducing the noise variables, the value of `estimator_status.gps_check_fail_flags` should be checked to ensure that it remains zero.

What should I do if the height estimate is diverging?

The most common cause of EKF height diverging away from GPS and altimeter measurements during flight is clipping and/or aliasing of the IMU measurements caused by vibration. If this is occurring, then the following signs should be evident in the data

- `ekf2_innovations.vel_pos_innov[3]` and `ekf2_innovations.vel_pos_innov[5]` will both have the same sign.
- `estimator_status.hgt_test_ratio` will be greater than 1.0

The recommended first step is to ensure that the autopilot is isolated from the airframe using an effective isolation mounting system. An isolator mount has 6 degrees of freedom, and therefore 6 resonant frequencies. As a general rule, the 6 resonant frequencies of the autopilot on the isolation mount should be above 25Hz to avoid interaction with the autopilot dynamics and below the frequency of the motors.

An isolation mount can make vibration worse if the resonant frequencies coincide with motor or propeller blade passage frequencies.

The EKF can be made more resistant to vibration induced height divergence by making the following parameter changes:

- Double the value of the innovation gate for the primary height sensor. If using barometric height this is `EKF2_BARO_GATE`.
- Increase the value of `EKF2_ACC_NOISE` to 0.5 initially. If divergence is still occurring, increase in further increments of 0.1 but do not go above 1.0

Note that the effect of these changes will make the EKF more sensitive to errors in GPS vertical velocity and barometric pressure.

What should I do if the position estimate is diverging?

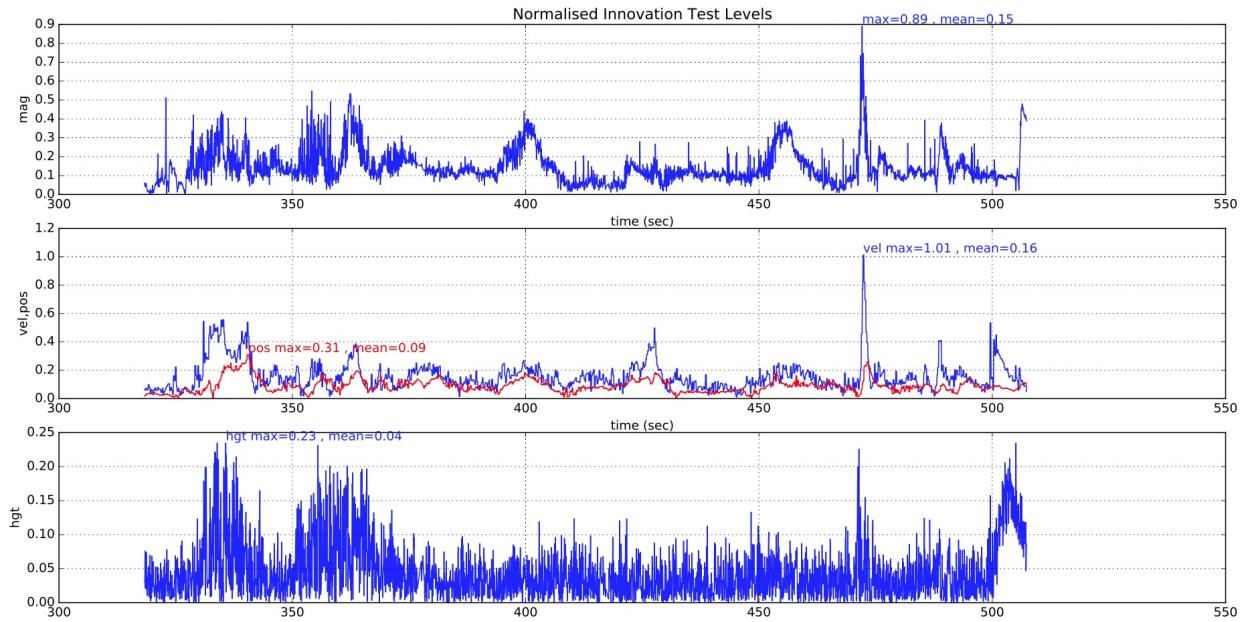
The most common causes of position divergence are:

- High vibration levels.
 - Fix by improving mechanical isolation of the autopilot.
 - Increasing the value of EKF2_ACC_NOISE and EKF2_GYR_NOISE can help, but does make the EKF more vulnerable to GPS glitches.
- Large gyro bias offsets.
 - Fix by re-calibrating the gyro. Check for excessive temperature sensitivity (> 3 deg/sec bias change during warm-up from a cold start and replace the sensor if affected or insulate to slow the rate of temeprature change.
- Bad yaw alignment
 - Check the magntometer calibration and alignment.
 - Check the heading shown QGC is within within 15 deg truth
- Poor GPS accuracy
 - Check for interference
 - Improve separation and shielding
 - Check flying location for GPS signal obstructions and reflectors (nearboy tall buildings)
- Loss of GPS

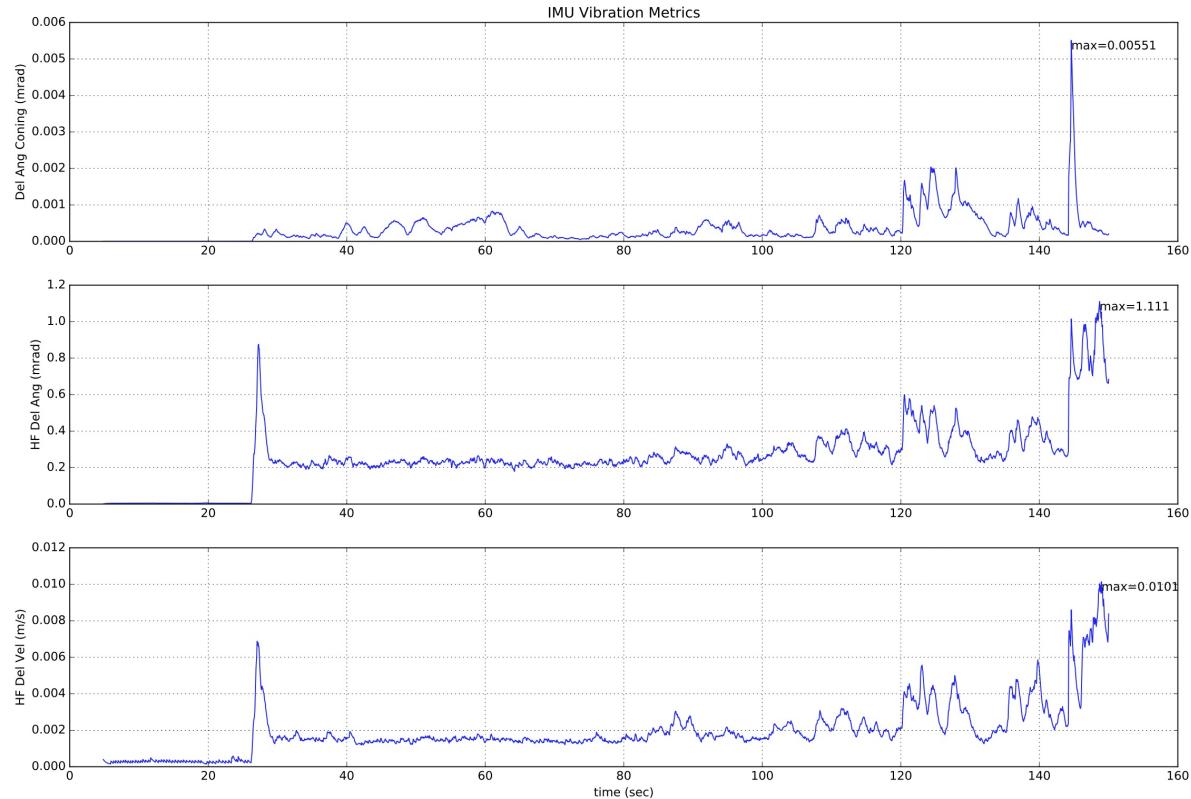
Determining which of these is the primary casue requires a methodical approach to analysis of the EKF log data:

- Plot the velcoity innovation test ratio - `estimator_status.vel_test_ratio`
- Plot the horizontal position innovation test ratio - `estimator_status.pos_test_ratio`
- Plot the height innovation test ratio - `estimator_status.hgt_test_ratio`
- Plot the magnetoemrer innovation test ratio - `estimator_status.mag_test_ratio`
- Plot the GPS receier reported speed accuracy - `vehicle_gps_position.s_variance_m_s`
- Plot the IMU delta angle state estimates - `estimator_status.states[10]`, `states[11]` and `states[12]`
- Plot the EKF internal high frequency vibration metrics:
 - Delta angle coning vibration - `estimator_status.vibe[0]`
 - High frequency delta angle vibration - `estimator_status.vibe[1]`
 - High frequency delta velocity vibration - `estimator_status.vibe[2]`

During normal operation, all the test ratios should remain below 0.5 with only occasional spikes above this as shown in the example below from a successful flight:



The following plot shows the EKF vibration metrics for a multirotor with good isolation. The landing shock and the increased vibration during takeoff and landing can be seen. Insufficient data has been gathered with these metrics to provide specific advice on maximum thresholds.



The above vibration metrics are of limited value as the presence of vibration at a frequency close to the IMU sampling frequency (1kHz for most boards) will cause offsets to appear in

the data that do not show up in the high frequency vibration metrics. The only way to detect aliasing errors is in their effect on inertial navigation accuracy and the rise in innovation levels.

In addition to generating large position and velocity test ratios of > 1.0, the different error mechanisms affect the other test ratios in different ways:

Determination of Excessive Vibration

High vibration levels normally affect vertical position and velocity innovations as well as the horizontal components. Magnetometer test levels are only affected to a small extent.

(insert example plots showing bad vibration here)

Determination of Excessive Gyro Bias

Large gyro bias offsets are normally characterised by a change in the value of delta angle bias greater than 5E-4 during flight (equivalent to ~3 deg/sec) and can also cause a large increase in the magnetometer test ratio if the yaw axis is affected. Height is normally unaffected other than extreme cases. Switch on bias value of up to 5 deg/sec can be tolerated provided the filter is given time to settle before flying. Pre-flight checks performed by the commander should prevent arming if the position is diverging.

(insert example plots showing bad gyro bias here)

Determination of Poor Yaw Accuracy

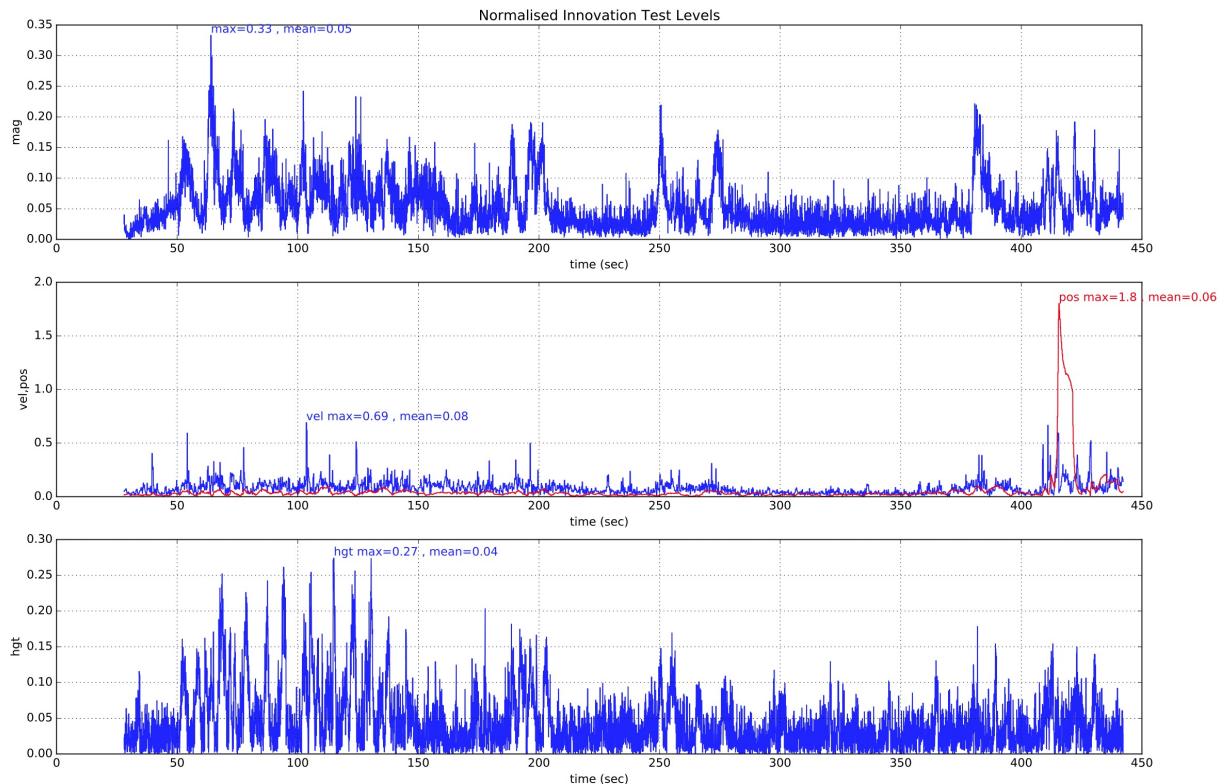
Bad yaw alignment causes a velocity test ratio that increases rapidly when the vehicle starts moving due to inconsistency in the direction of velocity calculated by the inertial nav and the GPS measurement. Magnetometer innovations are slightly affected. Height is normally unaffected.

(insert example plots showing bad yaw alignment here)

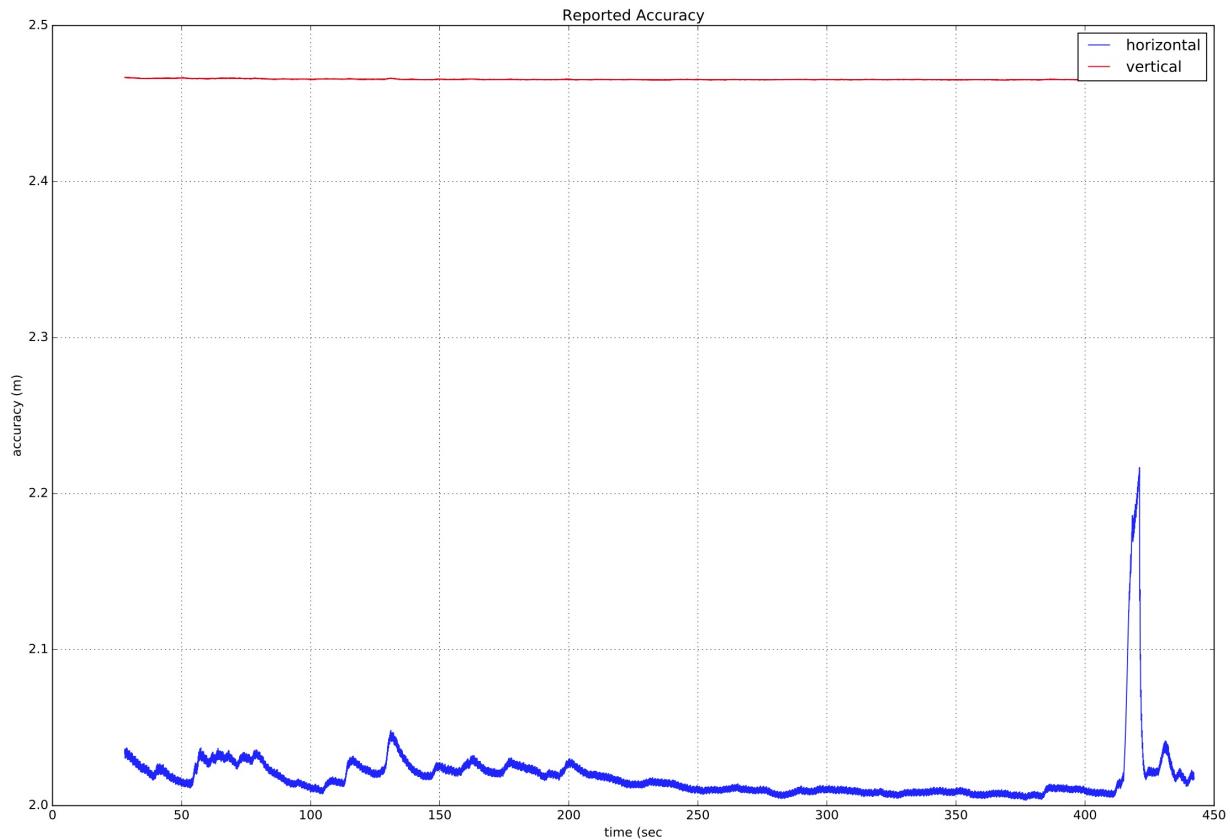
Determination of Poor GPS Accuracy

Poor GPS accuracy is normally accompanied by a rise in the reported velocity error of the receiver in conjunction with a rise in innovations. Transient errors due to multipath, obscuration and interference are more common causes. Here is an example of a temporary loss of GPS accuracy where the multi-rotor started drifting away from its loiter location and

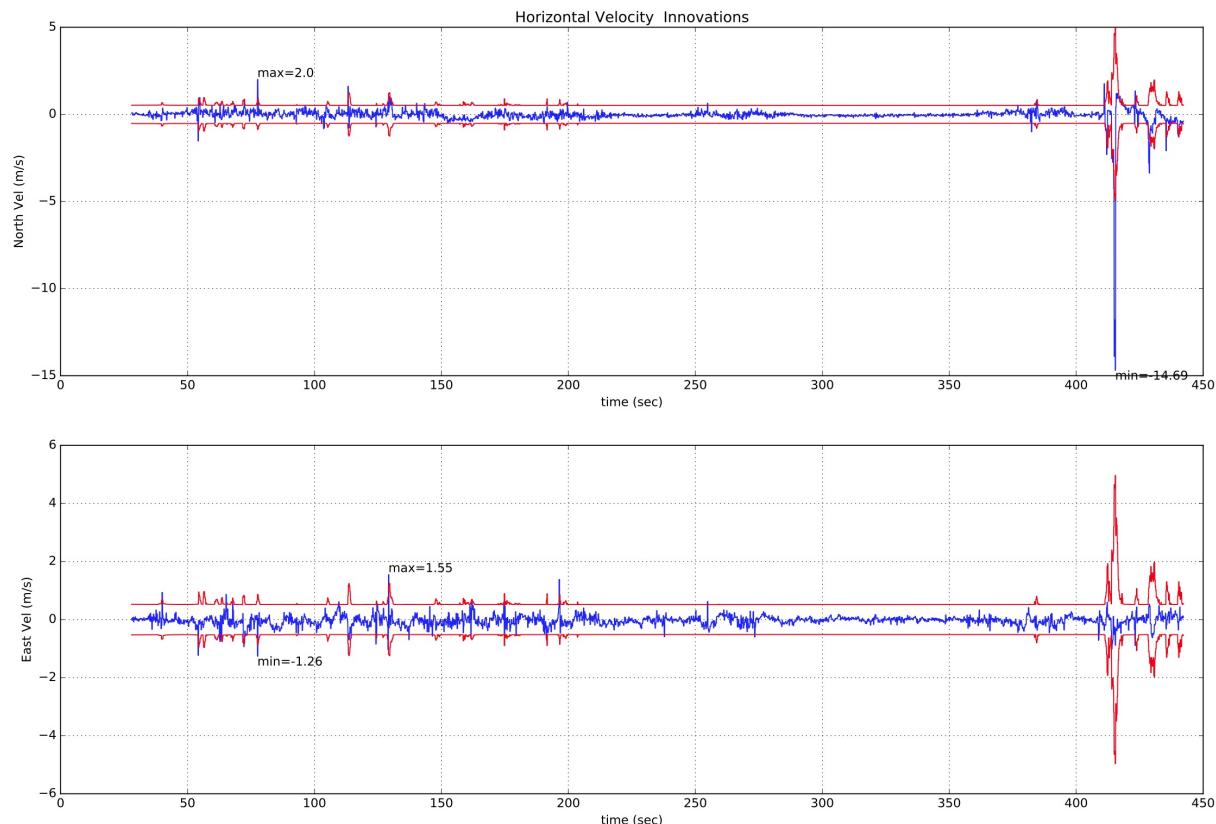
had to be corrected using the sticks. The rise in `estimator_status.vel_test_ratio` to greater than 1 indicates the GPs velocity was inconsistent with other measurements and has been rejected.



This is accompanied with rise in the GPS receivers reported velocity accuracy which indicates that it was likely a GPS error.



If we also look at the GPS horizontal velocity innovations and innovation variances, we can see the large spike in North velocity innovation that accompanies this GPS 'glitch' event.



Determination of GPS Data Loss

Loss of GPS data will be shown by the velocity and position innovation test ratios 'flat-lining'. If this occurs, check the other GPS status data in vehicle_gps_position for further information.

(insert example plots showing loss of GPS data here)

Preflight Sensor and EKF Checks

The commander module performs a number of preflight sensor quality and EKF checks which are controlled by the COM_ARM<> parameters. If these checks fail, the motors are prevented from arming and the following error messages are produced:

- PREFLIGHT FAIL: EKF HGT ERROR
 - This error is produced when the IMU and height measurement data are inconsistent.
 - Perform an accel and gyro calibration and restart the vehicle. If the error persists, check the height sensor data for problems.
 - The check is controlled by the COM_ARM_EKF_HGT parameter.
- PREFLIGHT FAIL: EKF VEL ERROR
 - This error is produced when the IMU and GPS velocity measurement data are inconsistent.
 - Check the GPS velocity data for un-realistic data jumps. If GPS quality looks OK, perform an accel and gyro calibration and restart the vehicle.
 - The check is controlled by the COM_ARM_EKF_VEL parameter.
- PREFLIGHT FAIL: EKF HORIZ POS ERROR
 - This error is produced when the IMU and position measurement data (either GPS or external vision) are inconsistent.
 - Check the position sensor data for un-realistic data jumps. If data quality looks OK, perform an accel and gyro calibration and restart the vehicle.
 - The check is controlled by the COM_ARM_EKF_POS parameter.
- PREFLIGHT FAIL: EKF YAW ERROR
 - This error is produced when the yaw angle estimated using gyro data and the yaw angle from the magnetometer or external vision system are inconsistent.
 - Check the IMU data for large yaw rate offsets and check the magnetometer alignment and calibration.
 - The check is controlled by the COM_ARM_EKF_POS parameter
- PREFLIGHT FAIL: EKF HIGH IMU ACCEL BIAS
 - This error is produced when the IMU accelerometer bias estimated by the EKF is excessive.
 - The check is controlled by the COM_ARM_EKF_AB parameter.
- PREFLIGHT FAIL: EKF HIGH IMU GYRO BIAS
 - This error is produced when the IMU gyro bias estimated by the EKF is excessive.
 - The check is controlled by the COM_ARM_EKF_GB parameter.
- PREFLIGHT FAIL: ACCEL SENSORS INCONSISTENT - CHECK CALIBRATION
 - This error message is produced when the acceleration measurements from

- different IMU units are inconsistent.
- This check only applies to boards with more than one IMU.
 - The check is controlled by the COM_ARM_IMU_ACC parameter.
- PREFLIGHT FAIL: GYRO SENSORS INCONSISTENT - CHECK CALIBRATION
 - This error message is produced when the angular rate measurements from different IMU units are inconsistent.
 - This check only applies to boards with more than one IMU.
 - The check is controlled by the COM_ARM_IMU_GYR parameter.

COM_ARM_WO_GPS

The COM_ARM_WO_GPS parameter controls whether arming is allowed without a GPS signal. This parameter must be set to 0 to allow arming when there is no GPS signal present. Arming without GPS is only allowed if the flight mode selected does not require GPS.

COM_ARM_EKF_POS

The COM_ARM_EKF_POS parameter controls the maximum allowed inconsistency between the EKF inertial measurements and position reference (GPS or external vision). The default value of 0.5 allows the differences to be no more than 50% of the maximum tolerated by the EKF and provides some margin for error increase when flight commences.

COM_ARM_EKF_VEL

The COM_ARM_EKF_VEL parameter controls the maximum allowed inconsistency between the EKF inertial measurements and GPS velocity measurements. The default value of 0.5 allows the differences to be no more than 50% of the maximum tolerated by the EKF and provides some margin for error increase when flight commences.

COM_ARM_EKF_HGT

The COM_ARM_EKF_HGT parameter controls the maximum allowed inconsistency between the EKF inertial measurements and height measurement (Baro, GPS, range finder or external vision). The default value of 0.5 allows the differences to be no more than 50% of the maximum tolerated by the EKF and provides some margin for error increase when flight commences.

COM_ARM_EKF_YAW

The COM_ARM_EKF_YAW parameter controls the maximum allowed inconsistency between the EKF inertial measurements and yaw measurement (magnetometer or external vision). The default value of 0.5 allows the differences to be no more than 50% of the maximum tolerated by the EKF and provides some margin for error increase when flight commences.

COM_ARM_EKF_AB

The COM_ARM_EKF_AB parameter controls the maximum allowed EKF estimated IMU accelerometer bias. The default value of 0.005 allows for up to 0.5 m/s/s of accelerometer bias.

COM_ARM_EKF_GB

The COM_ARM_EKF_GB parameter controls the maximum allowed EKF estimated IMU gyro bias. The default value of 0.00087 allows for up to 5 deg/sec of switch on gyro bias.

COM_ARM_IMU_ACC

The COM_ARM_IMU_ACC parameter controls the maximum allowed inconsistency in acceleration measurements between the default IMU used for flight control and other IMU units if fitted.

COM_ARM_IMU_GYR

The COM_ARM_IMU_GYR parameter controls the maximum allowed inconsistency in angular rate measurements between the default IMU used for flight control and other IMU units if fitted.

Software in the Loop (SITL) Simulation

Software in the Loop Simulation runs the complete system on the host machine and simulates the autopilot. It connects via local network to the simulator. The setup looks like this:



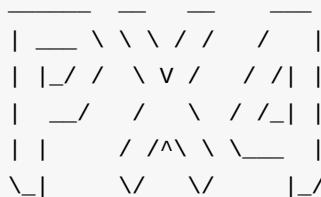
Running SITL

After ensuring that the [simulation prerequisites](#) are installed on the system, just launch: The convenience make target will compile the POSIX host build and run the simulation.

```
make posix_sitl_default jmavsim
```

This will bring up the PX4 shell:

```
[init] shell id: 140735313310464
[init] task name: px4
```



```
Ready to fly.
```

```
pxh>
```

Important Files

- The startup script is in the [posix-configs/SITL/init](#) folder and named `rcs_SIM_AIRFRAME`, the default is `rcs_jmavsim_iris`.
- The root file system (the equivalent of `/` as seen by the) is located inside the build directory: `build_posix_sitl_default/src/firmware/posix/rootfs/`

Taking it to the Sky

And a window with the 3D view of the [jMAVSim](#) simulator:



The system will print the home position once it finished initializing (`telem> home: 55.7533950, 37.6254270, -0.00`). You can bring it into the air by typing:

```
pxh> commander takeoff
```

Info Joystick or thumb-joystick support is available through QGroundControl (QGC). To use manual input, put the system in a manual flight mode (e.g. POSCTL, position control). Enable the thumb joystick from the QGC preferences menu.

Simulating a Wifi Drone

There is a special target to simulate a drone connected via Wifi on the local network:

```
make broadcast jmavsim
```

The simulator broadcasts his address on the local network as a real drone would do.

Extending and Customizing

To extend or customize the simulation interface, edit the files in the `Tools/jMAVSim` folder. The code can be accessed through the [jMAVSim repository](#) on Github.

Info The build system enforces the correct submodule to be checked out for all dependencies, including the simulator. It will not overwrite changes in files in the directory, however, when these changes are committed the submodule needs to be registered in the Firmware repo with the new commit hash. To do so, `git add Tools/jMAVSim` and commit the change. This will update the GIT hash of the simulator.

Interfacing to ROS

The simulation can be [interfaced to ROS](#) the same way as onboard a real vehicle.

Gazebo Simulation

Gazebo is a 3D simulation environment for autonomous robots. It supports standalone use (without ROS) or SITL + ROS.



[Video link](#)



Installation

The installation requires to install Gazebo and our simulation plugin.

Note Gazebo version 7 is recommended (the minimum version is Gazebo 6). If you run Linux and installed a ROS version earlier than Jade, be sure to uninstall the bundled Gazebo (`sudo apt-get remove ros-indigo-gazebo`) version as it is too old.

Mac OS

Mac OS requires Gazebo 7 which in turn requires xquartz and doesn't run without OpenCV.

```
brew cask install xquartz
brew install homebrew/science/opencv
brew install gazebo7
```

Linux

The PX4 SITL uses the Gazebo simulator, but does not depend on ROS. The simulation can be [interfaced to ROS](#) the same way as normal flight code is.

ROS Users

If you plan to use PX4 with ROS, make sure to follow the [Gazebo version guide for version 7](#) for ROS.

Normal Installation

Follow the [Linux installation instructions](#) for Gazebo 7.

Make sure to have both installed: `gazebo7` and `libgazebo7-dev`.

Running the Simulation

From within the source directory of the PX4 Firmware run the PX4 SITL with one of the airframes (Quads, planes and VTOL are supported, including optical flow):

Note You can use the instructions below to keep Gazebo running and only re-launch PX4.

Quadrotor

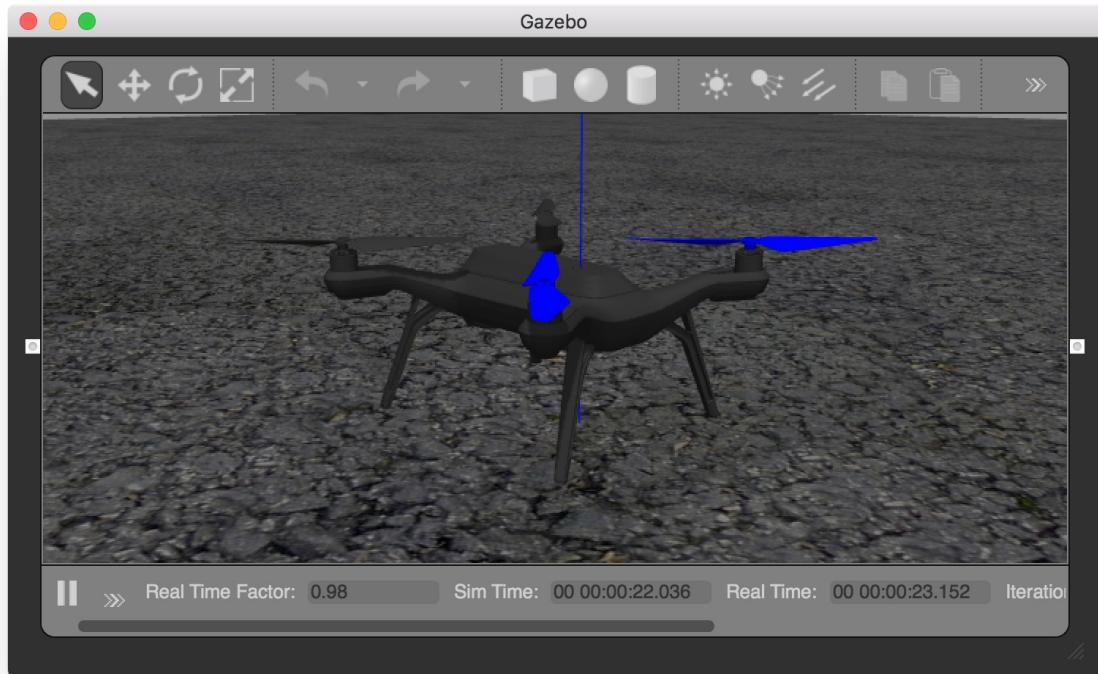
```
cd ~/src/Firmware
make posix_sitl_default gazebo
```

Quadrotor with Optical Flow

```
make posix_gazebo_iris_opt_flow
```

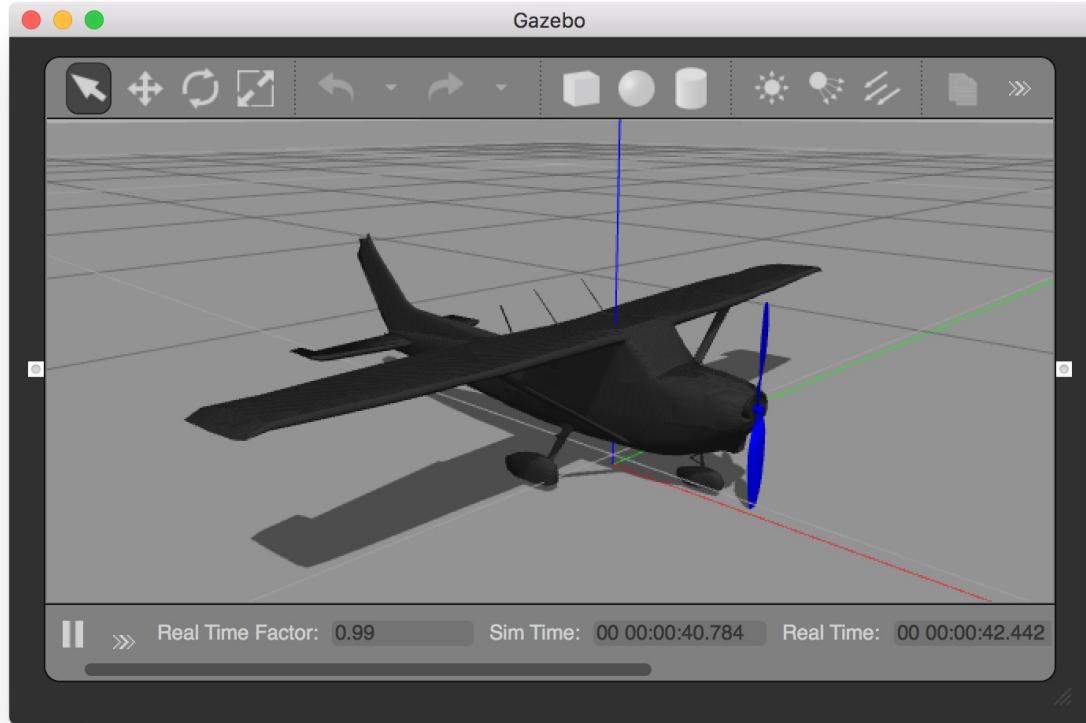
3DR Solo

```
make posix gazebo_solo
```



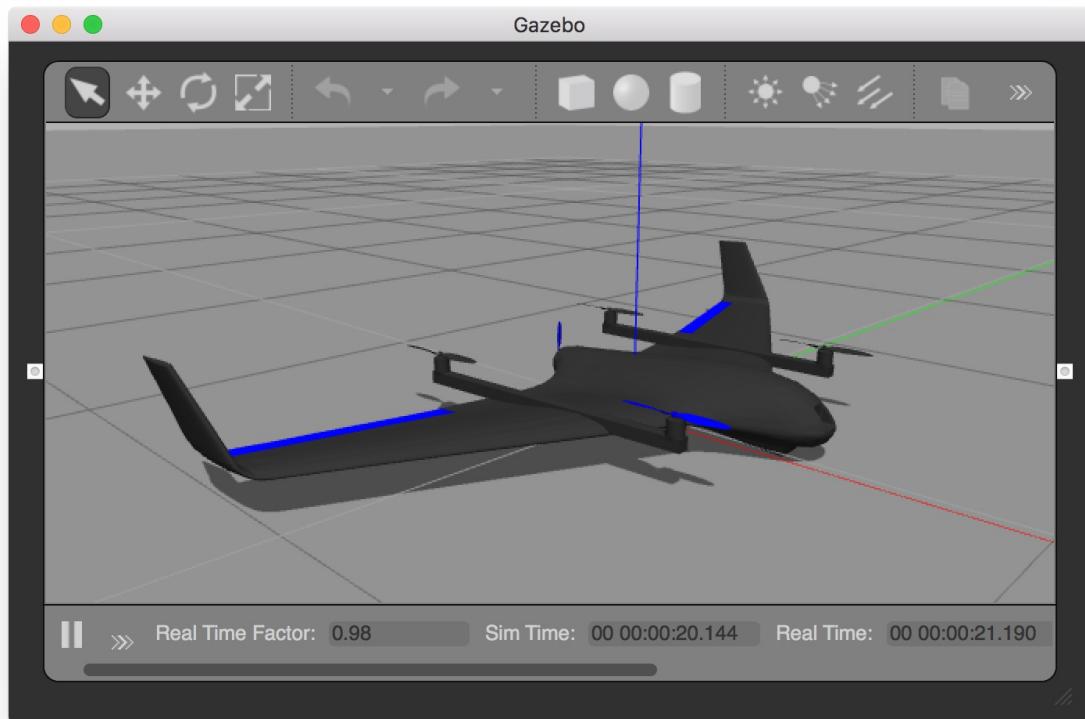
Standard Plane

```
make posix gazebo_plane
```



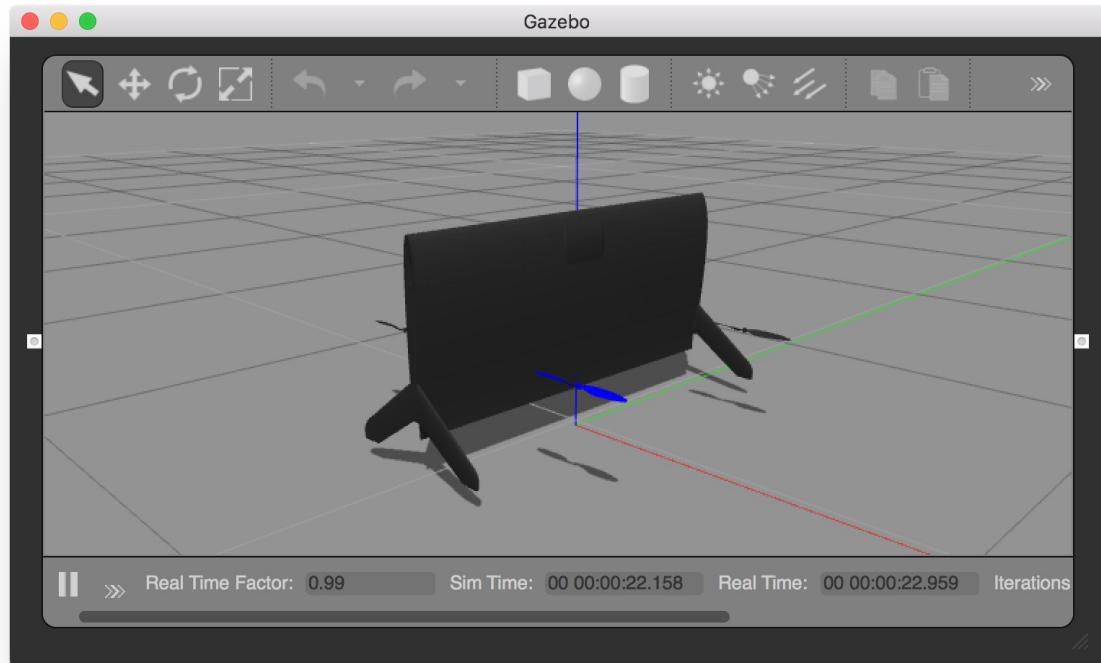
Standard VTOL

```
make posix_sitl_default gazebo_standard_vtol
```



Tailsitter VTOL

```
make posix_sitl_default gazebo_tailsitter
```



Change World

The current default world is the `iris.world` located in the directory [worlds](#). The default surrounding in the `iris.world` uses a heightmap as ground. This ground can cause difficulty when using a distance sensor. If there are unexpected results with that heightmap, it is recommended to change the model in `iris.model` from `uneven_ground` to `asphalt_plane`.

Taking it to the Sky

Note Please refer to the [Installing Files and Code](#) guide in case you run into any errors.

This will bring up the PX4 shell:

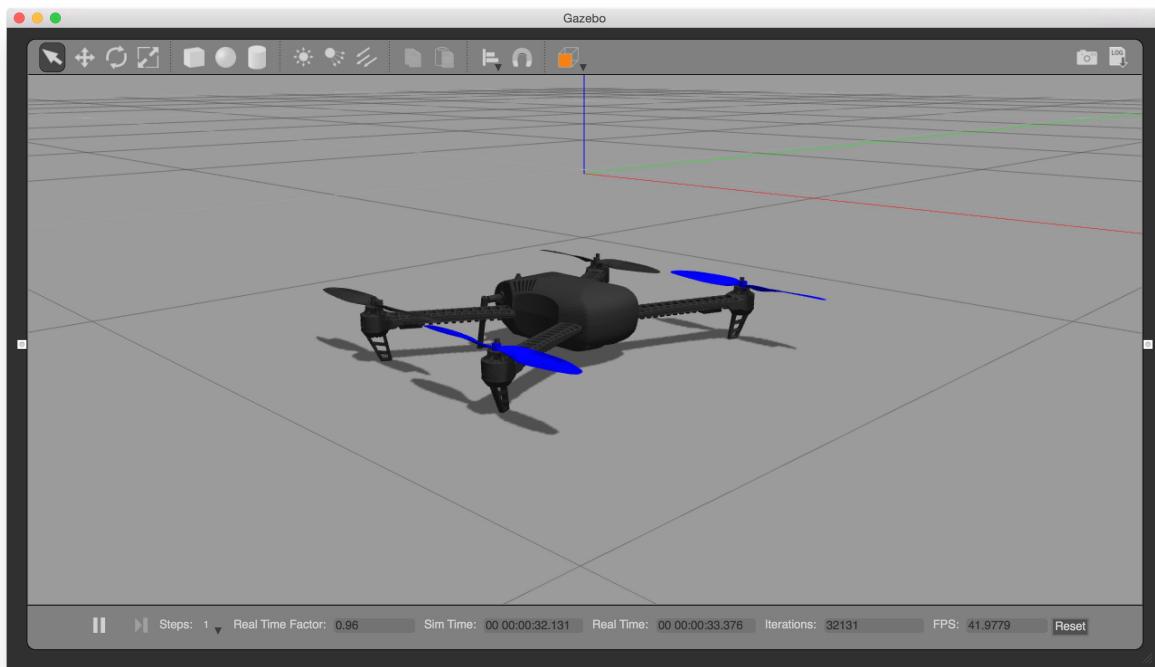
```
[init] shell id: 140735313310464
[init] task name: px4
```

```
____ \ \ \ / / / | |
| / \ \ v / / / | | |
| / / \ \ / / / | |
| | / / ^ \ \ \ | |
\| \ \ \ \ \ \ \ | /
```

px4 starting.

pxh>

Note Right-clicking the quadrotor model allows to enable follow mode from the context menu, which is handy to keep it in view.



The system will print the home position once it finished initializing (`telem> home: 55.7533950, 37.6254270, -0.00`). You can bring it into the air by typing:

```
pxh> commander takeoff
```

Note Joystick or thumb-joystick support is available through QGroundControl (QGC). To use manual input, put the system in a manual flight mode (e.g. POSCTL, position control). Enable the thumb joystick from the QGC preferences menu.

Starting Gazebo and PX4 separately

For extended development sessions it might be more convenient to start Gazebo and PX4 separately or even from within an IDE.

In addition to the existing cmake targets that run `sitl_run.sh` with parameters for px4 to load the correct model it creates a launcher targets named `px4_<mode>` that is a thin wrapper around original sitl px4 app. This thin wrapper simply embeds app arguments like current working directories and the path to the model file.

How to use it

- Run gazebo (or any other sim) server and client viewers via the terminal:

```
make posix_sitl_default gazebo_none_ide
```

- In your IDE select `px4_<mode>` target you want to debug (e.g. `px4_iris`)
- Start the debug session directly from IDE This approach significantly reduces the debug cycle time because simulator (e.g. gazebo) is always running in background and you only re-run the px4 process which is very light.

Extending and Customizing

To extend or customize the simulation interface, edit the files in the `Tools/sitl_gazebo` folder. The code is available on the [sitl_gazebo repository](#) on Github.

Note The build system enforces the correct GIT submodules, including the simulator. It will not overwrite changes in files in the directory.

Interfacing to ROS

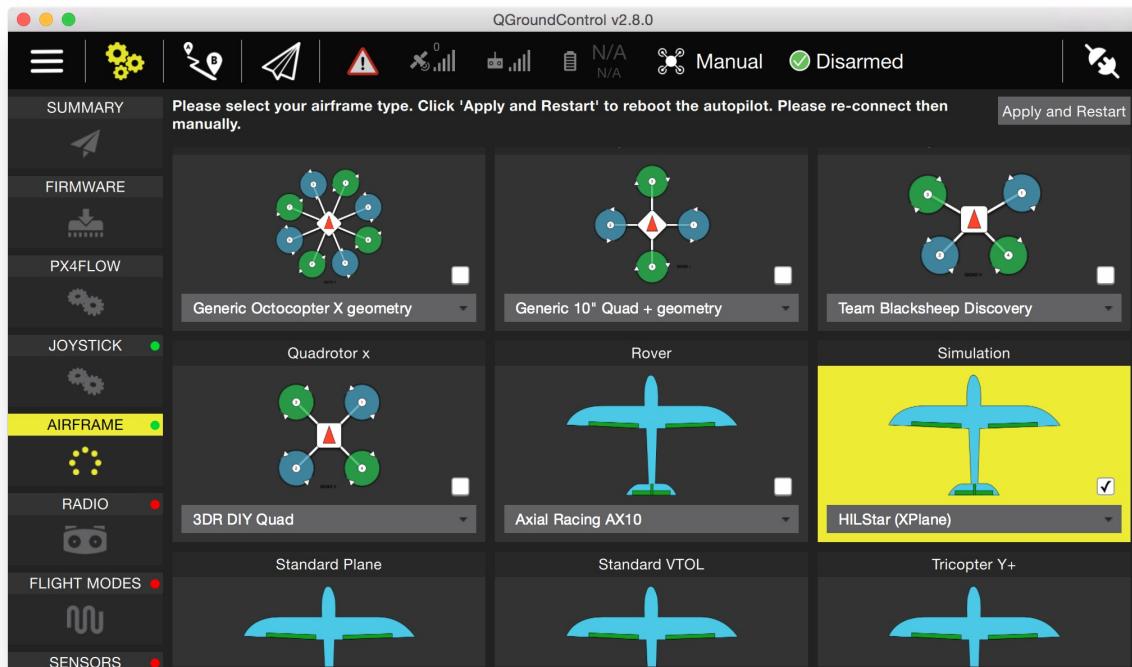
The simulation can be [interfaced to ROS](#) the same way as onboard a real vehicle.

Hardware in the Loop Simulation (HITL)

Hardware in the loop simulation is a simulation mode where the autopilot is connected to the simulator and all flight code runs on the autopilot. This approach has the benefit of testing the actual flight code on the real processor.

Configure the System for HITL

PX4 supports HITL for multicopters (using jMAVSIM) and fixed wing (using X-Plane demo or full). Flightgear support is present as well, but we generally recommend X-Plane. To enable it, configure it via the airframe menu.



Using jMAVSIM (Quadrotor)

- Make sure QGroundControl is not running (or accessing the device via serial port)
- Run jMAVSIM in HITL mode (replace the serial port if necessary):

```
./Tools/jmavsim_run.sh -q -d /dev/ttyACM0 -b 921600
```

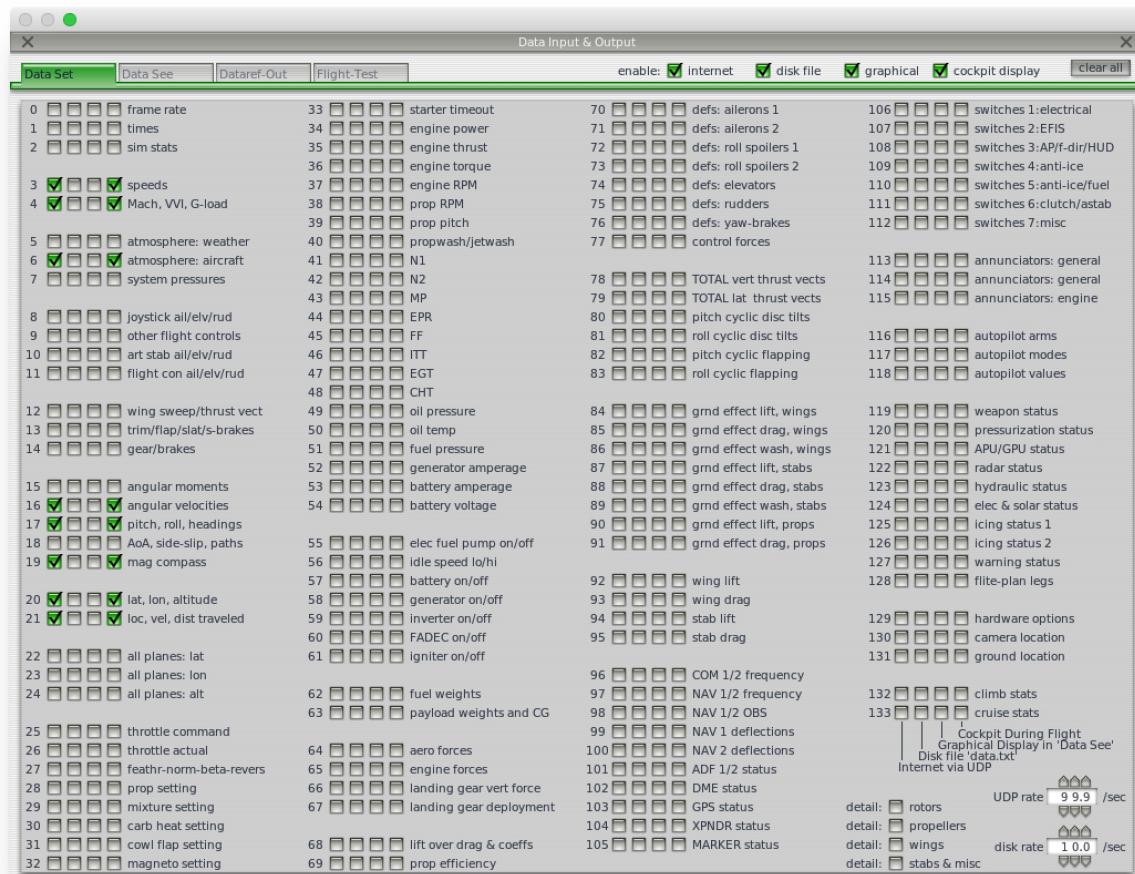
- The console will display mavlink text messages from the autopilot.

- Then run QGroundControl and connect via default UDP configuration.

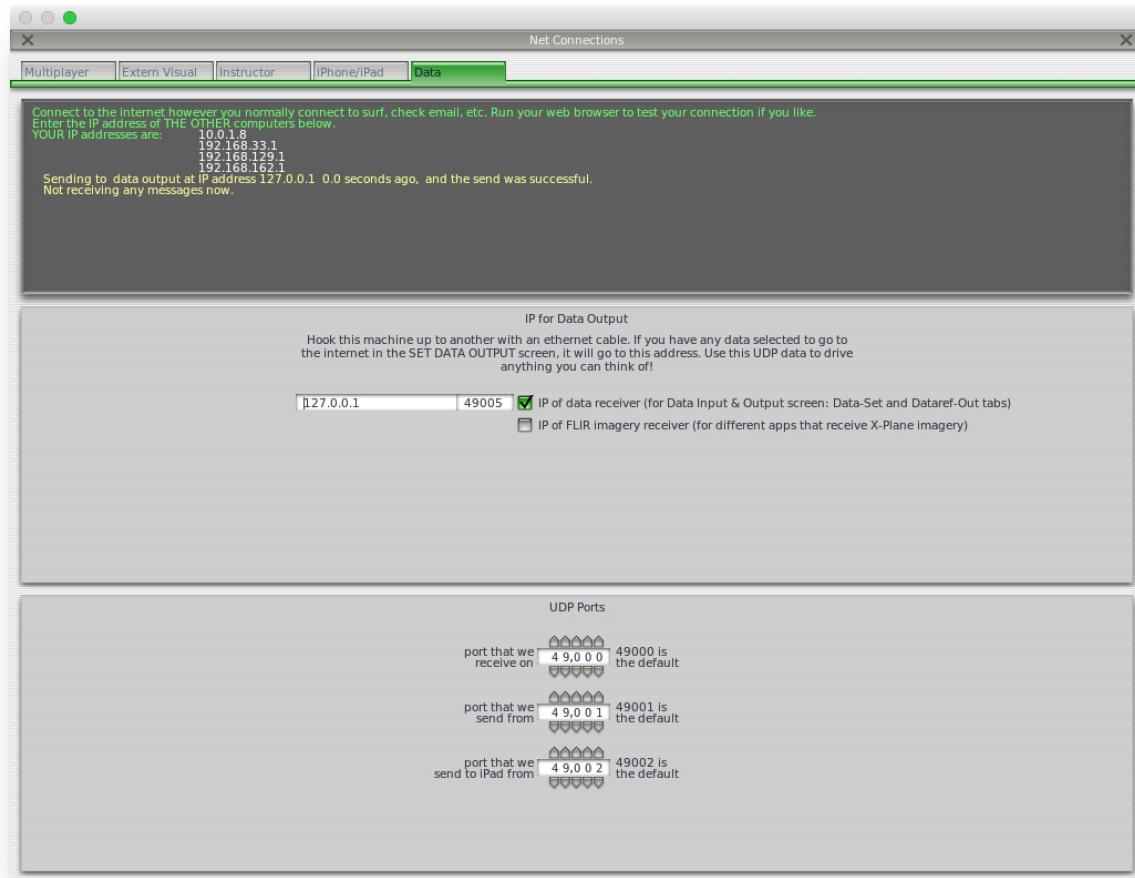
Using X-Plane

Enable Remote Access in X-Plane

In X-Plane two key settings have to be made: In Settings -> Data Input and Output, set these checkboxes:

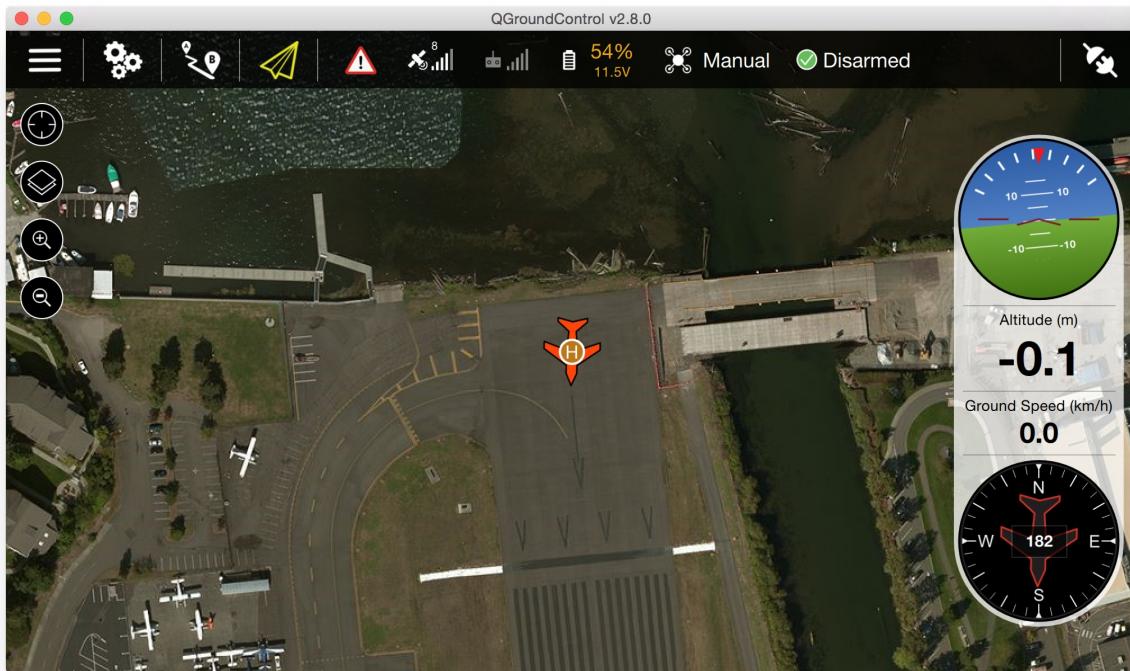


In Settings -> Net Connections in the Data tab, set localhost and port 49005 as IP address, as shown in the screenshot below:



Enable HITL in QGroundControl

Widgets -> HIL Config, then select X-Plane 10 in the drop-down and hit connect. Once the system is connected, battery status, GPS status and aircraft position should all become valid:



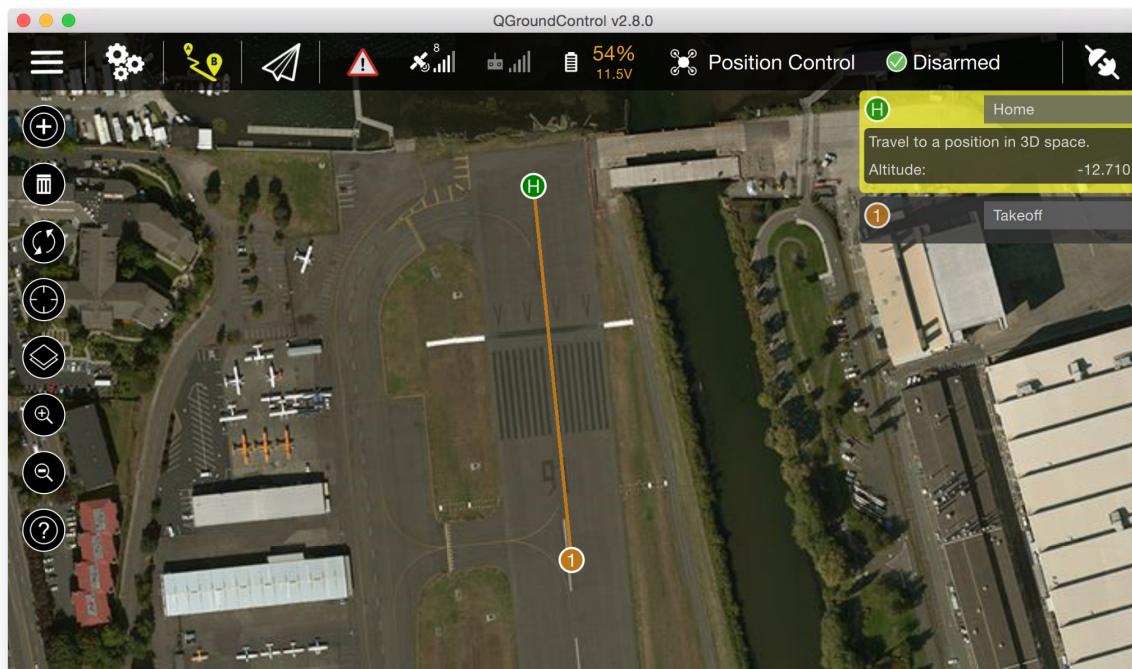
Switch to Joystick Input

If a joystick is preferred over a radio remote control, set the parameter `COM_RC_IN_MODE` to `1`. It can be found in the Commander parameter group.

Fly an Autonomous Mission in HITL

Switch to the flight planning view and put a single waypoint in front of the plane. Click on the sync icon to send the waypoint.

Then select MISSION from the flight mode menu in the toolbar and click on DISARMED to arm the plane. It will take off and loiter around the takeoff waypoint.



Interfacing the Simulation to ROS

The simulated autopilot starts a second MAVLink interface on port 14557. Connecting MAVROS to this port allows to receive all data the vehicle would expose if in real flight.

Launching MAVROS

If an interface to ROS is wanted, the already running secondary MAVLink instance can be connected to ROS via `mavros`. To connect to a specific IP (`fcu_url` is the IP / port of SITL), use a URL in this form:

```
roslaunch mavros px4.launch fcu_url:="udp://:14540@192.168.1.36:14557"
```

To connect to localhost, use this URL:

```
roslaunch mavros px4.launch fcu_url:="udp://:14540@127.0.0.1:14557"
```

Installing Gazebo for ROS

The Gazebo ROS SITL simulation is known to work with both Gazebo 6 and Gazebo 7, which can be installed via:

```
sudo apt-get install ros-$(ROS_DISTRO)-gazebo7-ros-pkgs //Recommended
```

or

```
sudo apt-get install ros-$(ROS_DISTRO)-gazebo6-ros-pkgs
```

Launching Gazebo with ROS wrappers

In case you would like to modify the Gazebo simulation to integrate sensors publishing directly to ROS topics e.g. the Gazebo ROS laser plugin, Gazebo must be launched with the appropriate ROS wrappers.

There are ROS launch scripts available to run the simulation wrapped in ROS:

- [posix_sitl.launch](#): plain SITL launch
- [mavros_posix_sitl.launch](#): SITL and MAVROS

To run SITL wrapped in ROS the ROS environment needs to be updated, then launch as usual:

(optional): only source the catkin workspace if you compiled MAVROS or other ROS packages from source

```
cd <Firmware_clone>
make posix_sitl_default gazebo
source ~/catkin_ws/devel/setup.bash      // (optional)
source Tools/setup_gazebo.bash $(pwd) $(pwd)/build_posix_sitl_default
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$(pwd)
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$(pwd)/Tools/sitl_gazebo
roslaunch px4 posix_sitl.launch
```

Include one of the above mentioned launch files in your own launch file to run your ROS application in the simulation.

What's happening behind the scenes

(or how to run it manually)

```
no_sim=1 make posix_sitl_default gazebo
```

This should start the simulator and the console will look like this

```
[init] shell id: 46979166467136
[init] task name: px4

____ \ \ \ / / / \ |
|_-/_ / \ \ / / / | | |
|_-/_ / \ \ / / / | |
| | / / ^ \ \ \ \_ | |
\_| \ \ \ \ \ \ \ \ \_ | /|
```

Ready to fly.

```
INFO LED::init
729 DevObj::init led
736 Added driver 0x2aba34001080 /dev/led0
INFO LED::init
742 DevObj::init led
INFO Not using /dev/ttyACM0 for radio control input. Assuming joystick input via MAVLink.
INFO Waiting for initial data on UDP. Please start the flight simulator to proceed..
```

Now in a new terminal make sure you will be able to insert the Iris model through the Gazebo menus, to do this set your environment variables to include the appropriate `sitl_gazebo` folders.

```
cd <Firmware_clone>
source Tools/setup_gazebo.bash $(pwd) $(pwd)/build_posix_sitl_default
```

Now start Gazebo like you would when working with ROS and insert the Iris quadcopter model. Once the Iris is loaded it will automatically connect to the px4 app.

```
roslaunch gazebo_ros empty_world.launch world_name:=$(pwd)/Tools/sitl_gazebo/worlds/iris.world
```

Crazyflie 2.0

The Crazyflie line of micro quads was created by Bitcraze AB. An overview of the Crazyflie 2 (CF2) is here: <https://www.bitcraze.io/crazyflie-2/>



Quick Summary

The main hardware documentation is here:

<https://wiki.bitcraze.io/projects:crazyflie2:index>

- Main System-on-Chip: STM32F405RG
 - CPU: 168 MHz ARM Cortex M4 with single-precision FPU
 - RAM: 192 KB SRAM
- nRF51822 radio and power management MCU
- MPU9250 Accel / Gyro / Mag
- LPS25H barometer

Flashing

After setting up the PX4 development environment, follow these steps to put the PX4 software on the CF2:

1. Grab source code of the PX4 [Bootloader](#)
2. Compile using `make crazyflie_bl`
3. Put the CF2 into DFU mode:
 - Ensure it is initially unpowered
 - Hold down button
 - Plug into computer's USB port
 - After a second, the blue LED should start blinking and after 5 seconds should start blinking faster
 - Release button
4. Flash bootloader using dfu-util: `sudo dfu-util -d 0483:df11 -a 0 -s 0x08000000 -D crazyflie_bl.bin` and unplug CF2 when done
 - If successful, then the yellow LED should blink when plugging in again
5. Grab the [Firmware](#)
6. Compile with `make crazyflie_default upload`
7. When prompted to plug in device, plug in CF2: the yellow LED should start blinking indicating bootloader mode. Then the red LED should turn on indicating that the flashing process has started.
8. Wait for completion
9. Done! Calibrate via QGC

Wireless

The onboard nRF module allows connecting to the board via Bluetooth or through the proprietary 2.4GHz Nordic ESB protocol.

- A [Crazyradio PA](#) is recommended.
- To fly the CF2 right away, the Crazyflie phone app is supported via Bluetooth

Using the official Bitcraze **Crazyflie phone app**

- Connect via Bluetooth
- Change mode in settings to 1 or 2
- Calibrate via QGC

Connecting via **MAVLink**

- Use a Crazyradio PA alongside a compatible GCS
- See [cfbridge](#) for how to connect any UDP capable GCS to the radio

Flying



[Video link](#)

Intel Aero

The Aero is a UAV development platform. Part of this is the Intel Aero Compute Board (see below), running Linux on a Quad-core CPU. This is connected to the FMU, which runs PX4 on NuttX.



Introduction

The main documentation is under <https://github.com/intel-aero/meta-intel-aero/wiki>. It includes instructions how to setup, update and connect to the board. And it also explains how to do development on the Linux side.

The following describes how to flash and connect to the FMU.

Flashing

After setting up the PX4 development environment, follow these steps to put the PX4 software on the FMU board:

1. Do a full update of all software on the Aero (<https://github.com/intel-aero/meta-intel-aero/wiki/Upgrade-To-Latest-Software-Release>)

2. Grab the [Firmware](#)
3. Compile with `make aerofc-v1_default`
4. Set the hostname (The following IP assumes you are connected via WiFi):

```
export AERO_HOSTNAME=192.168.1.1`
```

5. Upload with `make aerofc-v1_default upload`

Connecting QGroundControl via Network

1. Make sure you are connected to the board with WiFi or USB Network
2. ssh to the board and make sure mavlink forwarding runs. By default it automatically starts when booting. It can be started manually with:

```
/etc/init.d/mavlink_bridge.sh start
```

3. Start QGroundControl and it should automatically connect.
4. Instead of starting QGroundControl, you can open a [NuttX shell](#) with:

```
./Tools/mavlink_shell.py 0.0.0.0:14550
```

Pixfalcon Hardware

Pixfalcon is binary-compatible derivative of the [Pixhawk](#) design optimized for space-constrained applications such as FPV racers, designed by [Holybro](#). It has less IO to allow for the reduction in size. For drones requiring high processing performance or a camera interface the [Snapdragon Flight](#) might be a more optimal fit.



Quick Summary

- Main System-on-Chip: [STM32F427](#)
 - CPU: 180 MHz ARM Cortex M4 with single-precision FPU
 - RAM: 256 KB SRAM (L1)
- Failsafe System-on-Chip: STM32F100
 - CPU: 24 MHz ARM Cortex M3
 - RAM: 8 KB SRAM
- GPS: U-Blox M8 (bundled)
- Optical flow: PX4 Flow unit from manufacturer [Holybro](#) or distributor [Hobbyking](#)
- Availability: From manufacturer: [Holybro](#) or from distributor [Hobbyking](#)
- Digital Airspeed sensor from manufacturer [Holybro](#) or distributor [Hobbyking](#)
- On screen display with integrated Telemetry:
 - [Hobbyking OSD + US Telemetry \(915 MHz\)](#)
 - [Hobbyking OSD + EU Telemetry \(433 MHz\)](#)
- Pure Telemetry options:
 - [Hobbyking Wifi Telemetry](#)
 - [Hobbyking EU Micro Telemetry \(433 MHz\)](#)

- [Hobbyking US Micro Telemetry \(915 MHz\)](#)

Connectivity

- 1x I2C
- 2x UART (one for Telemetry / OSD, no flow control)
- 8x PWM with manual override
- S.BUS / PPM input

Pixhawk Hardware

[Pixhawk](#) is the standard microcontroller platform for the PX4 flight stack. It runs the PX4 Middleware on the [NuttX](#) OS. As a CC-BY-SA 3.0 licensed Open Hardware design, all schematics and design files are [available](#). The [Pixfalcon](#) is a smaller version of Pixhawk for FPV racers and similar platforms. For drones requiring high processing performance or a camera interface the [Snapdragon Flight](#) might be a more optimal fit.



Quick Summary

- Main System-on-Chip: [STM32F427](#)
 - CPU: 180 MHz ARM Cortex M4 with single-precision FPU
 - RAM: 256 KB SRAM (L1)
- Failsafe System-on-Chip: STM32F100
 - CPU: 24 MHz ARM Cortex M3
 - RAM: 8 KB SRAM
- Wifi: ESP8266 external
- GPS: U-Blox 7/8 (Hobbyking) / U-Blox 6 (3D Robotics)
- Optical flow: [PX4 Flow unit](#)
- Availability:
 - [Hobbyking EU version \(433 MHz\)](#)
 - [Hobbyking US version \(915 MHz\)](#)
 - [3D Robotics Store](#) (GPS and telemetry not bundled)
- Accessories:
 - [Digital airspeed sensor](#)

- [Hobbyking Wifi Telemetry](#)
- [Hobbyking OSD + US Telemetry \(915 MHz\)](#)
- [Hobbyking OSD + EU Telemetry \(433 MHz\)](#)

Connectivity

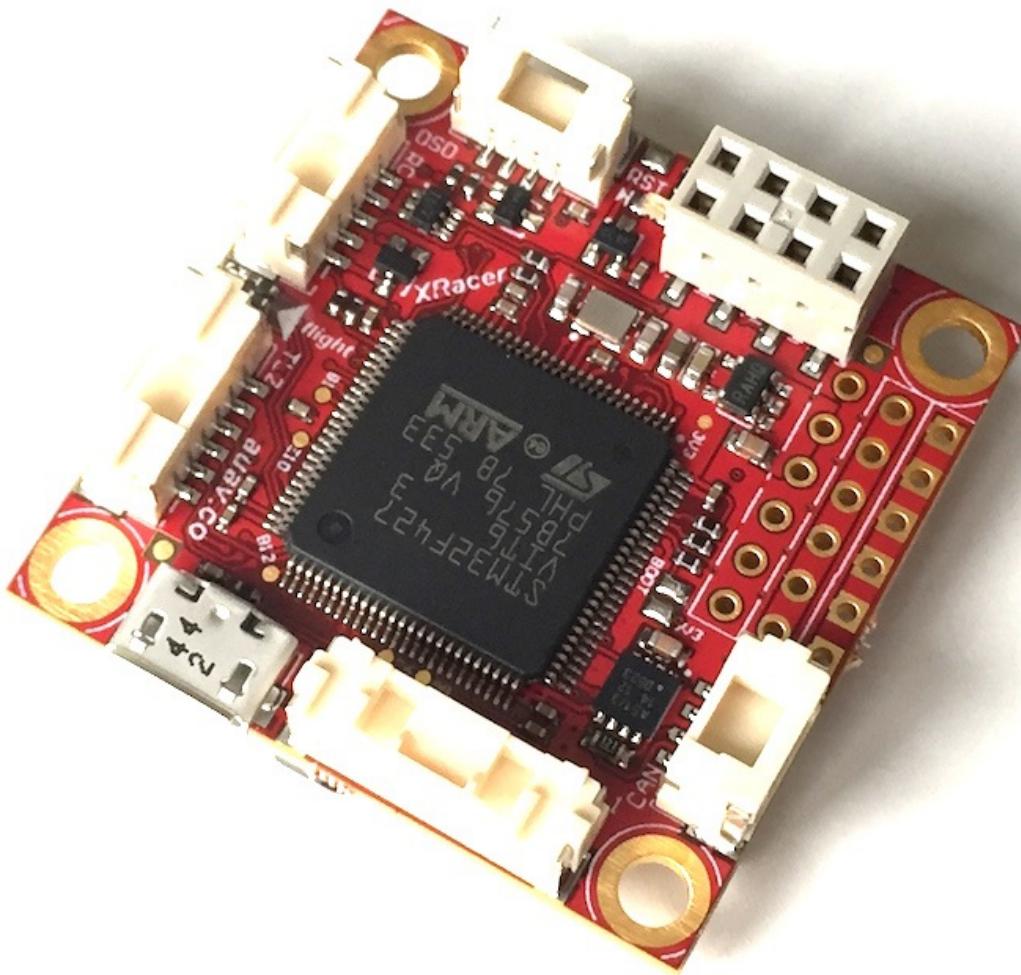
- 1x I2C
- 1x CAN (2x optional)
- 1x ADC
- 4x UART (2x with flow control)
- 1x Console
- 8x PWM with manual override
- 6x PWM / GPIO / PWM input
- S.BUS / PPM / Spektrum input
- S.BUS output

Pinouts and Schematics

The board is documented in detailed on the [Pixhawk project](#) website.

Pixracer

The Pixhawk XRacer board family is optimized for small racing quads and planes. In contrast to [Pixfalcon](#) and [Pixhawk](#) it has in-built Wifi, new sensors, convenient full servo headers, CAN and supports 2M flash.



Quick Summary

The main hardware documentation is here: <https://pixhawk.org/modules/pixracer>

- Main System-on-Chip: [STM32F427VIT6 rev.3](#)
 - CPU: 180 MHz ARM Cortex M4 with single-precision FPU
 - RAM: 256 KB SRAM (L1)
- Standard FPV form factor: 36x36 mm with standard 30.5 mm hole pattern
- Wifi telemetry and software upgrade

- Invensense ICM-20608 Accel / Gyro (4 KHz) / MPU9250 Accel / Gyro / Mag (4 KHz)
- HMC5983 magnetometer with temperature compensation
- Measurement Specialties MS5611 barometer
- JST GH connectors
- microSD (logging)
- S.BUS / Spektrum / SUMD / PPM input
- FrSky telemetry port
- OneShot PWM out (configurable)
- Optional: Safety switch and buzzer
- Availability:
 - [AUAV Pixracer](#)
- Accessories:
 - [Digital airspeed sensor](#)
 - [Hobbyking OSD + US Telemetry \(915 MHz\)](#)
 - [Hobbyking OSD + EU Telemetry \(433 MHz\)](#)

Kit

The Pixracer is designed to use a separate avionics power supply. This is necessary to avoid current surges from motors or ESCs to flow back to the flight controller and disturb its delicate sensors.

- Power module (with voltage and current sensing)
- I2C splitter (supporting AUAV, Hobbyking and 3DR peripherals)
- Cable kit for all common peripherals

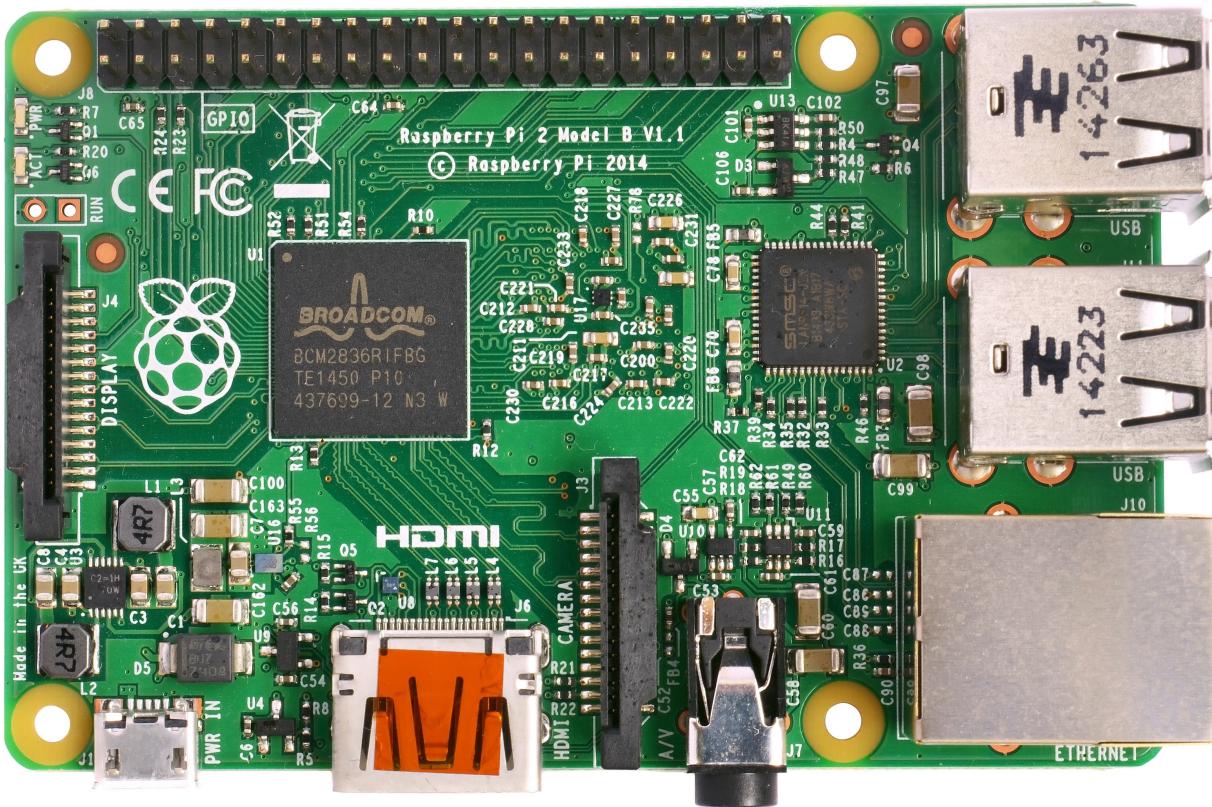
Wifi (no USB required)

One of the main features of the board is its ability to use Wifi for flashing new firmware, system setup and in-flight telemetry. This frees it of the need of any desktop system.

Setup and telemetry are already available, firmware upgrade is already supported by the default bootloader but not yet enabled

- [ESP8266 Documentation and Flash Instructions](#)
- [Custom ESP8266 MAVLink firmware](#)

Raspberry Pi 2/3 Autopilot



Developer Quick Start

OS Image

Use the [Emlid RT Raspbian image for Navio 2](#). The default image will have most of the setup procedures shown below already done.

Important: make sure not to upgrade the system (more specifically the kernel). By upgrading, a new kernel can get installed which lacks the necessary HW support (you can check with `ls /sys/class/pwm`, the directory should not be empty).

Setting up access

The Raspbian image has SSH setup already. Username is "pi" and password is "raspberry". You can connect to your RPi2/3 over a network (Ethernet is set to come up with DHCP by default) and then proceed to configure WiFi access. We assume that the username and password remain at their defaults for the purpose of this guide.

To setup the RPi2/3 to join your local wifi, follow [this guide](#).

Find the IP address of your Pi from your network, and then you can proceed to connect to it using SSH.

```
ssh pi@<IP-ADDRESS>
```

Expand the Filesystem

After installing the OS and connecting to it, make sure to [expand the Filesystem](#), so there is enough space on the SD Card.

Changing hostnames

To avoid conflicts with any other RPis on the network, we advise you to change the default hostname to something sensible. We used "px4autopilot" for our setup. Connect to the Pi via SSH and follow the below instructions.

Edit the hostname file:

```
sudo nano /etc/hostname
```

Change `raspberry` to whatever hostname you want (one word with limited characters apply)

Next you need to change the hosts file:

```
sudo nano /etc/hosts
```

Change the entry `127.0.1.1 raspberry` to `127.0.1.1 <YOURNEWHOSTNAME>`

Reboot the Pi after this step is completed to allow it to re-associate with your network.

Setting up Avahi (Zeroconf)

To make connecting to the Pi easier, we recommend setting up Avahi (Zeroconf) which allows easy access to the Pi from any network by directly specifying its hostname.

```
sudo apt-get install avahi-daemon  
sudo insserv avahi-daemon
```

Next, setup the Avahi configuration file

```
sudo nano /etc/avahi/services/multiple.service
```

Add this to the file :

```
<?xml version="1.0" standalone='no'?>
<!DOCTYPE service-group SYSTEM "avahi-service.dtd">
<service-group>
    <name replace-wildcards="yes">%h</name>
    <service>
        <type>_device-info._tcp</type>
        <port>0</port>
        <txt-record>model=RackMac</txt-record>
    </service>
    <service>
        <type>_ssh._tcp</type>
        <port>22</port>
    </service>
</service-group>
```

Restart the daemon

```
sudo /etc/init.d/avahi-daemon restart
```

And that's it. You should be able to access your Pi directly by its hostname from any computer on the network.

You might have to add .local to the hostname to discover it.

Configuring a SSH Public-Key

In order to allow the PX4 development environment to automatically push executables to your board, you need to configure passwordless access to the RPi. We use the public-key authentication method for this.

To generate new SSH keys enter the following commands (Choose a sensible hostname such as `<YOURNAME>@<YOURDEVICE>` . Here we have used `pi@px4autopilot`)

These commands need to be run on the HOST development computer!

```
ssh-keygen -t rsa -C pi@px4autopilot
```

Upon entering this command, you'll be asked where to save the key. We suggest you save it in the default location (`$HOME/.ssh/id_rsa`) by just hitting Enter.

Now you should see the files `id_rsa` and `id_rsa.pub` in your `.ssh` directory in your home folder:

```
ls ~/.ssh  
authorized_keys  id_rsa  id_rsa.pub  known_hosts
```

The `id_rsa` file is your private key. Keep this on the development computer. The `id_rsa.pub` file is your public key. This is what you put on the targets you want to connect to.

To copy your public key to your Raspberry Pi, use the following command to append the public key to your `authorized_keys` file on the Pi, sending it over SSH:

```
cat ~/.ssh/id_rsa.pub | ssh pi@px4autopilot 'cat >> .ssh/authorized_keys'
```

Note that this time you will have to authenticate with your password ("raspberry" by default).

Now try `ssh pi@px4autopilot` and you should connect without a password prompt.

If you see a message "Agent admitted failure to sign using the key." then add your RSA or DSA identities to the authentication agent, `ssh-agent` and then execute the following command:

```
ssh-add
```

If this did not work, delete your keys with `rm ~/.ssh/id*` and follow the instructions again.

Testing file transfer

We use SCP to transfer files from the development computer to the target board over a network (WiFi or Ethernet).

To test your setup, try pushing a file from the development PC to the Pi over the network now. Make sure the Pi has network access, and you can SSH into it.

```
echo "Hello" > hello.txt  
scp hello.txt pi@px4autopilot:/home/pi/  
rm hello.txt
```

This should copy over a "hello.txt" file into the home folder of your RPi. Validate that the file was indeed copied, and you can proceed to the next step.

Native builds (optional)

You can run PX4 builds directly on the Pi if you desire. This is the *native* build. The other option is to run builds on a development computer which cross-compiles for the Pi, and pushes the PX4 executable binary directly to the Pi. This is the *cross-compiler* build, and the recommended one for developers due to speed of deployment and ease of use.

For cross-compiling setups, you can skip this step.

The steps below will setup the build system on the Pi to that required by PX4. Run these commands on the Pi itself!

```
sudo apt-get update  
sudo apt-get install cmake python-empy
```

Then clone the Firmware directly onto the Pi.

Building the code

Continue with our [standard build system installation](#).

Snapdragon Flight Autopilot

The Snapdragon Flight platform is a high-end autopilot / onboard computer which runs the PX4 Flight Stack on the DSP on the QuRT real time operating system using the [DSPAL API](#) for POSIX compatibility. In comparison to [Pixhawk](#) it adds a camera and WiFi and high-end processing power, and different IO.

More information about the Snapdragon Flight platform is at [Snapdragon-Flight-Details](#)



Quick Summary

- System-on-Chip: [Snapdragon 801](#)
 - CPU: Quad-core 2.26 GHz Krait
 - DSP: Hexagon DSP (QDSP6 V5A) – 801 MHz+256KL2 (running the flight code)
 - GPU: Qualcomm® Adreno™ 330 GPU
 - RAM: 2GB LPDDR3 PoP @931 MHz
- Storage: 32GB eMMC Flash
- Video: Sony IMX135 on Liteon Module 12P1BAD11
 - 4k@30fps 3840×2160 video capture to SD card with H.264 @ 100Mbits (1080p/60 with parallel FPV), 720p FPV
- Optic Flow: Omnivision OV7251 on Sunny Module MD102A-200
 - 640x480 @ 30/60/90 fps
- Wifi: Qualcomm® VIVE™ 1-stream 802.11n/ac with MU-MIMO † Integrated digital core
- BT/WiFi: BT 4.0 and 2G/5G WiFi via QCA6234
 - 802.11n, 2×2 MIMO with 2 uCOAX connectors on-board for connection to external antenna
- GPS: Telit Jupiter SE868 V2 module (use of an external u-Blox module is recommended by PX4 instead)
 - uCOAX connector on-board for connection to external GPS patch antenna

- CSR SiRFstarV @ 5Hz via UART
- Accelerometer / Gyro / Mag: Invensense MPU-9250 9-Axis Sensor, 3x3mm QFN, on bus SPI1
- Baro: Bosch BMP280 barometric pressure sensor, on bus I2C3
- Power: 5VDC via external 2S-6S battery regulated down to 5V via APM adapter
- Availability: [Intrinsyc Store](#)

Connectivity

- One USB 3.0 OTG port (micro-A/B)
- Micro SD card slot
- Gimbal connector (PWB/GND/BLSP)
- ESC connector (2W UART)
- I2C
- 60-pin high speed Samtec QSH-030-01-L-D-A-K expansion connector
 - 2x BLSP ([BAM Low Speed Peripheral](#))
 - USB

Pinouts

Although the Snapdragon uses DF13 connectors, the pinout is different from Pixhawk.

WiFi

- WLAN0, WLAN1 (+BT 4.0): U.FL connector: [Taoglas adhesive antenna \(DigiKey\)](#)

Connectors

The default mapping of the serial ports is as follows:

Device	Description
/dev/tty-1	J15 (next to USB)
/dev/tty-2	J13 (next to power module connector)
/dev/tty-3	J12 (next to J13)
/dev/tty-4	J9 (next to J15)

For a custom UART to BAM mapping, create a file called "blsp.config" and adb push it to `/usr/share/data/adsp`. E.g., to keep the default mapping, your "blsp.config" should look as follows:

```
tty-1 bam-9 2-wire
tty-2 bam-8 2-wire
tty-3 bam-6 2-wire
tty-4 bam-2 2-wire
```

Be sure to include the text "2-wire" at the end of each line to allow the UART to use only the TX and RX pins specified in the tables below. If 2-wire is not specified (or if the file is not present on the target) the UART will default to using 4-wire mode and will require an additional two pins for RTS/CTS flow control. This will cause a problem for any other type of I/O on the same connector, since the pins will be configured as RTS and CTS signals. If, for example, J9 (described below) was being used to connect to both a UART and an I2C device, the I2C signals on pin 4 and pin 6 would be configured as RTS and CTS signals, overriding the I2C SDA and SCL signals.

J9 / GPS

Pin	2-wire UART + I2C	4-wire UART	SPI	Comment
1	3.3V	3.3V	3.3V	
2	UART2_TX	UART2_TX	SPI2_MOSI	Output (3.3V)
3	UART2_RX	UART2_RX	SPI2_MISO	Input (3.3V)
4	I2C2_SDA	UART2_RTS	SPI2_CS	(3.3V)
5	GND	GND	GND	
6	I2C2_SCL	UART2_CTS	SPI2_CLK	(3.3V)

J12 / Gimbal bus

Pin	2-wire UART + GPIO	4-wire UART	SPI	Comment
1	3.3V	3.3V	3.3V	
2	UART8_TX	UART8_TX	SPI8_MOSI	Output (3.3V)
3	UART8_RX	UART8_RX	SPI8_MISO	Input (3.3V)
4	APQ_GPIO_47	UART8_RTS	SPI8_CS	(3.3V)
5	GND	GND	GND	
6	APQ_GPIO_48	UART8_CTS	SPI8_CLK	(3.3V)

J13 / ESC bus

Pin	2-wire UART + GPIO	4-wire UART	SPI	Comment
1	5V	5V	5V	
2	UART6_TX	UART6_TX	SPI6_MOSI	Output (5V)
3	UART6_RX	UART6_RX	SPI6_MISO	Input (5V)
4	APQ_GPIO_29	UART6_RTS	SPI6_CS	(5V)
5	GND	GND	GND	
6	APQ_GPIO_30	UART6_CTS	SPI6_CLK	(5V)

J14 / Power

Pin	Signal	Comment
1	5V DC	Power input
2	GND	
3	I2C3_SCL	(5V)
4	I2C3_SDA	(5V)

J15 / Radio Receiver / Sensors

Pin	2-wire UART + I2C	4-wire UART	SPI	Comment
1	3.3V	3.3V	3.3V	
2	UART9_TX	UART9_TX	SPI9_MOSI	Output
3	UART9_RX	UART9_RX	SPI9_MISO	Input
4	I2C9_SDA	UART9_RTS	SPI9_CS	
5	GND	GND	GND	
6	I2C9_SCL	UART9_CTS	SPI9_CLK	

Peripherals

UART to Pixracer / Pixfalcon Wiring

This interface is used to leverage the Pixracer / Pixfalcon as I/O interface board. Connect to `TELEM1` Pixfalcon and to `TELEM2` on Pixracer.

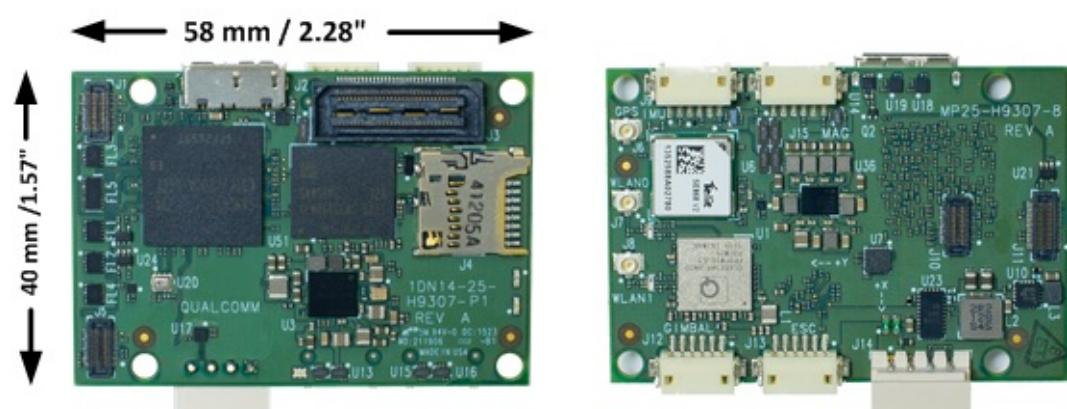
Snapdragon J13 Pin	Signal	Comment	Pixfalcon / Pixracer Pin
1	5V	Power for autopilot	5V
2	UART6_TX	Output (5V) TX -> RX	3
3	UART6_RX	Input (5V) RX -> TX	2
4	APQ_GPIO_29	(5V)	Not connected
5	GND		6
6	APQ_GPIO_30	(5V)	Not connected

GPS Wiring

Even though the 3DR GPS is described to have a 5v input, operation with 3.3V seems to work fine. (The built-in regulator MIC5205 has a minimum operating voltage of 2.5v.)

Snapdragon J9 Pin	Signal	Comment	3DR GPS 6pin/4pin	Pixfalcon GPS pin	3DR PIXHAWK MINI GPS
1	3.3V	(3.3V)	1	4	3 (5V)
2	UART2_TX	Output (3.3V)	2/-	3	4
3	UART2_RX	Input (3.3V)	3/-	2	5
4	I2C2_SDA	(3.3V)	-/3	5	2
5	GND		6/-	1	6
6	I2C2_SCL	(3.3V)	-/2	6	1

Dimensions



Optical Flow on the Snapdragon Flight

The Snapdragon Flight board has a downward facing gray-scale camera which can be used for optical flow based position stabilization.

Besides a camera, optical flow requires a downward facing distance sensor. Here, the use of the TeraRanger One is discussed.

TeraRanger One setup

To connect the TeraRanger One (TROne) to the Snapdragon Flight, the TROne I2C adapter must be used. The TROne must be flashed with the I2C firmware by the vendor.

The TROne is connected to the Snapdragon Flight through a custom DF13 4-to-6 pin cable. The wiring is as follows:

4 pin	<->	6 pin
1		1
2		6
3		4
4		5

The TROne must be powered with 10 - 20V.

Optical flow

The optical flow is computed on the application processor and sent to PX4 through Mavlink. Clone and compile the [snap_cam](#) repo according to the instructions in its readme.

Run the optical flow application as root:

```
optical_flow -n 50 -f 30
```

The optical flow application requires IMU Mavlink messages from PX4. You may have to add an additional Mavlink instance to PX4 by adding the following to your `px4.config`:

```
mavlink start -u 14557 -r 1000000 -t 127.0.0.1 -o 14558  
mavlink stream -u 14557 -s HIGHRES_IMU -r 250
```

Snapdragon Advanced

Connect to Snapdragon

Over FTDI

Connect the small debug header shipped with the Snapdragon and the FTDI cable.

On Linux, open a console using:

```
screen /dev/ttyUSB0 115200
```

Change USB0 to whatever it happens to be. Check `/dev/` or `/dev/serial/by-id`.

Over ADB (Android Debug Bridge)

Connect the Snapdragon over USB2.0 and power it up using the power module. When the Snapdragon is running the, the LED will be slowly blinking (breathing) in blue.

Make sure the board can be found using adb:

```
adb devices
```

If you cannot see the device, it is most likely a USB device permission issue. Follow the instructions

To get a shell, do:

```
adb shell
```

Upgrade Snapdragon

For this step the Flight_BSP zip file from Intrynsic is required. It can be obtained after registering using the board serial.

Upgrading/replacing the Linux image

Flashing the Linux image will erase everything on the Snapdragon. Back up your work

before you perform this step!

Make sure the board can be found using adb:

```
adb devices
```

Then, reboot it into the fastboot bootloader:

```
adb reboot bootloader
```

Make sure the board can be found using fastboot:

```
fastboot devices
```

Download the latest BSP from Intrinsyc:

```
unzip Flight_3.1.1_BSP_apq8074-00003.zip  
cd BSP/binaries/Flight_BSP_4.0  
../fastboot-all.sh
```

It is normal that the partitions `recovery`, `update`, and `factory` will fail.

Updating the ADSP firmware

Part of the PX4 stack is running on the ADSP (the DSP side of the Snapdragon 8074). The underlying operating system QURT needs to be updated separately.

If anything goes wrong during the ADSP firmware update, your Snapdragon can get bricked!

Follow the steps below carefully which should prevent bricking in most cases.

First of all, if you're not already on BSP 3.1.1, [upgrade the Linux image!](#)

Prevent bricking

To prevent the system from hanging on boot because of anything wrong with the ADSP firmware, do the following changes before updating:

Edit the file directly on the Snapdragon over `screen` or `adb shell`:

```
vim /usr/local/qr-linux/q6-admin.sh
```

Or load the file locally and edit it there with the editor of your choice:

To do this, load the file locally:

```
adb pull /usr/local/qr-linux/q6-admin.sh
```

Edit it:

```
gedit q6-admin.sh
```

And push it back:

```
adb push q6-admin.sh /usr/local/qr-linux/q6-admin.sh  
adb shell chmod +x /usr/local/qr-linux/q6-admin.sh
```

Comment out the while loops causing boot to hang:

```
# Wait for adsp.mdt to show up  
#while [ ! -s /lib/firmware/adsp.mdt ]; do  
#  sleep 0.1  
#done
```

and:

```
# Don't leave until ADSP is up  
#while [ "`cat /sys/kernel/debug/msm_subsys/adsp`" != "2" ]; do  
#  sleep 0.1  
#done
```

Push the latest ADSP firmware files

Download the file [Flight_3.1.1a_qcom_flight_controller_hexagon_sdk_add_on.zip](#) from Intrinsyc.

And copy them on to the Snapdragon:

```
unzip Flight_3.1.1a_qcom_flight_controller_hexagon_sdk_add_on.zip  
cd images/8074-eagle/normal/adsp_proc/obj/qdsp6v5_ReleaseG/LA/system/etc/firmware  
adb push . /lib/firmware
```

Then do a graceful reboot, so that the firmware gets applied:

```
adb reboot
```

Serial ports

Use serial ports

Not all POSIX calls are currently supported on QURT. Therefore, some custom ioctl are needed.

The APIs to set up and use the UART are described in [dspal](#).

Wifi-settings

These are notes for advanced developers.

Connect to the Linux shell (see [console instructions](#)).

Access point mode

If you want the Snapdragon to be a wifi access point (AP mode), edit the file:

`/etc/hostapd.conf` and set:

```
ssid=EnterYourSSID  
wpa_passphrase=EnterYourPassphrase
```

Then configure AP mode:

```
/usr/local/qr-linux/wificonfig.sh -s softap  
reboot
```

Station mode

If you want the Snapdragon to connect to your existing wifi, edit the file:

`/etc/wpa_supplicant/wpa_supplicant.conf` and add your network settings:

```
network={  
    ssid="my existing network ssid"  
    psk="my existing password"  
}
```

Then configure station mode:

```
/usr/local/qr-linux/wificonfig.sh -s station  
reboot
```

Troubleshooting

adb does not work

- Check [permissions](#)
- Make sure you are using a working Micro-USB cable.
- Try a USB 2.0 port.
- Try front and back ports of your computer.

USB permissions

1) Create a new permissions file

```
sudo -i gedit /etc/udev/rules.d/51-android.rules
```

paste this content, which enables most known devices for ADB access:

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="0bb4", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0e79", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0502", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0b05", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="413c", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0489", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="091e", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="18d1", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0bb4", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="12d1", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="24e3", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="2116", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0482", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="17ef", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="1004", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="22b8", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0409", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="2080", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0955", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="2257", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="10a9", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="1d4d", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0471", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="04da", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="05c6", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="1f53", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="04e8", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="04dd", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0fce", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0930", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="19d2", MODE="0666", GROUP="plugdev"
```

Set up the right permissions for the file:

```
sudo chmod a+r /etc/udev/rules.d/51-android.rules
```

Restart the deamon

```
sudo udevadm control --reload-rules
sudo service udev restart
sudo udevadm trigger
```

If it still doesn't work, check [this answer on StackOverflow](#).

Board doesn't start / is boot-looping / is bricked

If you can still connect to the board using the serial console and get to a prompt such as:

```
root@linaro-developer:~#
```

You can get into fastboot (bootloader) mode by entering:

```
reboot2fastboot
```

If the serial console is not possible, you can try to connect the Micro USB cable, and enter:

```
adb wait-for-device && adb reboot bootloader
```

Then power cycle the board. If you're lucky, adb manages to connect briefly and can send the board into fastboot.

To check if it's in fastboot mode, use:

```
fastboot devices
```

Once you managed to get into fastboot mode, you can try [above steps](#) to update the Android/Linux image.

If you happen to have a [P2 board](#), you should be able to reset the Snapdragon to the recovery image by starting up the Snapdragon while shorting the two pins next to where J3 is written (The two rectangular pins in-between the corner hole and the MicroSD card slot almost at the edge of the board).

If everything fails, you probably need to request help from intrinsyc.

No space left on device

Sometimes `make eagle_default upload` fails to upload:

```
failed to copy 'px4' to '/home/linaro/px4': No space left on device
```

This can happen if ramdumps fill up the disk. To clean up, do:

```
rm -rf /var/log/ramdump/*
```

Also, the logs might have filled the space. To delete them, do:

```
rm -rf /root/log/*
```

Undefined PLT symbol

_FDtest

If you see the following output on mini-dm when trying to start the px4 program, it means that you need to [update the ADSP firmware](#):

```
[08500/03] 05:10.960 HAP:45:undefined PLT symbol _FDtest (689) /libpx4muorb_skel.so
0303 symbol.c
```

Something else

If you have changed the source, presumably added functions and you see `undefined PLT symbol ...` it means that the linking has failed.

- Do the declaration and definition of your function match one to one?
- Is your code actually getting compiled? Is the module listed in the [cmake config](#)?
- Is the (added) file included in the `CMakeLists.txt` ?
- Try adding it to the POSIX build and running the compilation. The POSIX linker will inform you about linking errors at compile/linking time.

krait update param XXX failed on startup

```
ERROR [platforms_posix_px4_layer] krait update param 297 failed
ERROR [platforms_posix_px4_layer] krait update param 646 failed

px4 starting.
ERROR [muorb] Initialize Error calling the uorb fastrpc initialize method..
ERROR [muorb] ERROR: FastRpcWrapper Not Initialized
```

If you get errors like the above when starting px4, try

- [upgrading the Linux image](#)
- and [updating the ADSP firmware](#). Also try to do this from a native Linux installation instead of a virtual machine. There have been [reports](#) where it didn't seem to work when done in a virtual machine.
- then [rebuild the px4 software](#), by first completely deleting your existing Firmware repo and then recloning it [as described here](#)
- and finally [rebuild and re-run it](#)
- make sure the executable bit of `/usr/local/qr-linux/q6-admin.sh` is set: `adb shell chmod +x /usr/local/qr-linux/q6-admin.sh`

ADSP restarts

If the mini-dm console suddenly shows a whole lot of INIT output, the ADSP side has crashed. The reasons for it are not obvious, e.g. it can be some segmentation fault, null pointer exception, etc..

The mini-dm console output typically looks like this:

```
[08500/02] 20:32.332 Process Sensor launched with ID=1 0130 main.c
[08500/02] 20:32.337 mmpm_register: MMPM client for USM ADSP core 12 0117 Ultrasou
ndStreamMgr_Mmpm.cpp
[08500/02] 20:32.338 ADSP License DB: License validation function with id 164678 sto
red. 0280 adsp_license_db.cpp
[08500/02] 20:32.338 AvsCoreSvc: StartSvcHandler Enter 0518 AdspCoreSvc.cpp
[08500/02] 20:32.338 AdspCoreSvc: Started successfully 0534 AdspCoreSvc.cpp
[08500/02] 20:32.342 DSPS INIT 0191 sns_init_dsps.c
[08500/02] 20:32.342 INIT DONE 0224 sns_init_dsps.c
[08500/02] 20:32.342 Sensors Init : waiting(1) 0160 sns_init_dsps.c
[08500/02] 20:32.342 INIT DONE 0224 sns_init_dsps.c
[08500/02] 20:32.342 THRD CREATE: Thread=0x39 Name(Hex)= 53, 4e, 53, 5f, 53, 4d, 47,
52 0186 qurt_elite_thread.cpp
[08500/02] 20:32.343 THRD CREATE: Thread=0x38 Name(Hex)= 53, 4e, 53, 5f, 53, 41, 4d,
0 0186 qurt_elite_thread.cpp
[08500/02] 20:32.343 THRD CREATE: Thread=0x37 Name(Hex)= 53, 4e, 53, 5f, 53, 43, 4d,
0 0186 qurt_elite_thread.cpp
[08500/02] 20:32.343 THRD CREATE: Thread=0x35 Name(Hex)= 53, 4e, 53, 5f, 50, 4d, 0,
0 0186 qurt_elite_thread.cpp
[08500/02] 20:32.343 THRD CREATE: Thread=0x34 Name(Hex)= 53, 4e, 53, 5f, 53, 53, 4d,
0 0186 qurt_elite_thread.cpp
[08500/02] 20:32.343 THRD CREATE: Thread=0x33 Name(Hex)= 53, 4e, 53, 5f, 44, 45, 42,
55 0186 qurt_elite_thread.cpp
[08500/02] 20:32.343 Sensors Init : ///////////init once completed/////////// 0169
sns_init_dsps.c
[08500/02] 20:32.342 loading BLSP configuration 0189 blsp_config.c
[08500/02] 20:32.343 Sensors DIAG F3 Trace Buffer Initialized 0260 sns_init_dsps.c
[08500/02] 20:32.345 INIT DONE 0224 sns_init_dsps.c
[00053/03] 20:32.345 Unsupported algorithm service id 0 0953 sns_scm_ext.c
[08500/02] 20:32.346 INIT DONE 0224 sns_init_dsps.c
[08500/02] 20:32.347 INIT DONE 0224 sns_init_dsps.c
[08500/02] 20:32.347 INIT DONE 0224 sns_init_dsps.c
[08500/02] 20:32.546 HAP:159:unable to open the specified file path 0167 file.c
[08500/04] 20:32.546 failed to open /usr/share/data/adsp/blsp.config 0204 blsp_con
fig.c
[08500/04] 20:32.546 QDSP6 Main.c: blsp_config_load() failed 0261 main.c
[08500/02] 20:32.546 Loaded default UART-BAM mapping 0035 blsp_config.c
[08500/02] 20:32.546 UART tty-1: BAM-9 0043 blsp_config.c
[08500/02] 20:32.546 UART tty-2: BAM-6 0043 blsp_config.c
[08500/02] 20:32.546 UART tty-3: BAM-8 0043 blsp_config.c
[08500/02] 20:32.546 UART tty-4: BAM-2 0043 blsp_config.c
[08500/02] 20:32.546 UART tty-5: BAM N/A 0048 blsp_config.c
[08500/02] 20:32.546 UART tty-6: BAM N/A 0048 blsp_config.c
[08500/02] 20:32.547 HAP:111:cannot find /oemconfig.so 0141 load.c
[08500/03] 20:32.547 HAP:4211::error: -1: 0 == dynconfig_init(&conf, "security") 0
696 sigverify.c
[08500/02] 20:32.548 HAP:76:cannot find /voiceproc_tx.so 0141 load.c
[08500/02] 20:32.550 HAP:76:cannot find /voiceproc_rx.so 0141 load.c
```

Do I have a P1 or P2 board?

The silkscreen on the Snapdragon reads something like:

```
1DN14-25-
H9550-P1
REV A
QUALCOMM
```

If you see **H9550**, it means you have a P2 board!

Please ignore that it says -P1.

Presumably P1 boards don't have a factory partition/image and therefore can't be restored to factory state.

Accessing I/O Data

Low level bus data can be accessed from code running on the aDSP, using a POSIX-like API called DSPAL. The header files for this API are maintained on [github](#) and are commented with Doxygen formatted documentation in each header file. A description of the API's supported and links to the applicable header files is provided below.

API Overview

- Serial: [qurt_serial.h](#)
- I2C: [qurt_i2c.h](#)
- SPI: [qurt_spi.h](#)
- GPIO: [qurt_gpio.h](#)
- Timers: [qurt_timer.h](#)
- Power Control: [HAP_power.h](#)

Sample Source Code

The unit test code to verify each DSPAL function also represent good examples for how to call the functions.

This code is also on [github](#)

Setting the Serial Data Rate

The serial API does not conform to the termios convention for setting data rate through the tcsetattr() function. IOCTL codes are used instead and are described in the header file linked above.

Timers

Additional functions for more advanced aDSP operations are available with the prefix qurt_. Timer functions, for example, are available with the qurt_timer prefix and are documented in the qurt_timer.h header file included with the [Hexagon SDK](#).

Setting the Power Level

Using the HAP functions provided by the Hexagon SDK, it is possible to set the power level of the aDSP. This will often lead to reduced I/O latencies. More information on these API's is available in the [HAP_power.h](#) header file available in the [Hexagon SDK](#).

Snapdragon: Camera and Optical Flow

Please follow the instructions in [this README](#) to use the camera and optical flow with a Snapdragon Flight.

uORB Messaging

Introduction

The uORB is an asynchronous publish() / subscribe() messaging API used for inter-thread/inter-process communication.

Look at the [tutorial](#) to learn how to use it in C++.

uORB is automatically started early on bootup as many applications depend on it. It is started with `uorb start`. Unit tests can be started with `uorb test`.

Adding a new topic

To add a new topic, you need to create a new `.msg` file in the `msg/` directory and add the file name to the `msg/CMakeLists.txt` list. From this, there will automatically be C/C++ code generated.

Have a look at the existing `.msg` files for supported types. A message can also be used nested in other messages. To each generated C/C++ struct, a field `uint64_t timestamp` will be added. This is used for the logger, so make sure to fill it in when logging the message.

To use the topic in the code, include the header:

```
#include <uORB/topics/topic_name.h>
```

By adding a line like the following in the `.msg` file, a single message definition can be used for multiple independent topic instances:

```
# TOPICS mission offboard_mission onboard_mission
```

Then in the code, use them as topic id: `ORB_ID(offboard_mission)`.

Publishing

Publishing a topic can be done from anywhere in the system, including interrupt context (functions called by the `hrt_call` API). However, advertising a topic is only possible outside of interrupt context. A topic has to be advertised in the same process as its later published.

Listing Topics and Listening in

The 'listener' command is only available on Pixracer (FMUv4) and Linux / OS X.

To list all topics, list the file handles:

```
ls /obj
```

To listen to the content of one topic for 5 messages, run the listener:

```
listener sensor_accel 5
```

The output is n-times the content of the topic:

```
TOPIC: sensor_accel #3
timestamp: 84978861
integral_dt: 4044
error_count: 0
x: -1
y: 2
z: 100
x_integral: -0
y_integral: 0
z_integral: 0
temperature: 46
range_m_s2: 78
scaling: 0
```

```
TOPIC: sensor_accel #4
timestamp: 85010833
integral_dt: 3980
error_count: 0
x: -1
y: 2
z: 100
x_integral: -0
y_integral: 0
z_integral: 0
temperature: 46
range_m_s2: 78
scaling: 0
```

MAVLink Messaging

An overview of all messages can be found [here](#).

Create Custom MAVLink Messages

This tutorial assumes you have a `custom uORB ca_trajectory message` in `msg/ca_trajectory.msg` and a custom mavlink `ca_trajectory message` in `mavlink/include/mavlink/v1.0/custom_messages/mavlink_msg_ca_trajectory.h` (see [here](#) how to create a custom mavlink message and header).

Sending Custom MAVLink Messages

This section explains how to use a custom uORB message and send it as a mavlink message.

Add the headers of the mavlink and uorb messages to `mavlink_messages.cpp`

```
#include <uORB/topics/ca_trajectory.h>
#include <v1.0/custom_messages/mavlink_msg_ca_trajectory.h>
```

Create a new class in `mavlink_messages.cpp`

```
class MavlinkStreamCaTrajectory : public MavlinkStream
{
public:
    const char *get_name() const
    {
        return MavlinkStreamCaTrajectory::get_name_static();
    }
    static const char *get_name_static()
    {
        return "CA_TRAJECTORY";
    }
    uint8_t get_id()
    {
        return MAVLINK_MSG_ID_CA_TRAJECTORY;
    }
    static MavlinkStream *new_instance(Mavlink *mavlink)
    {
        return new MavlinkStreamCaTrajectory(mavlink);
    }
}
```

```

        unsigned get_size()
    {
        return MAVLINK_MSG_ID_CA_TRAJECTORY_LEN + MAVLINK_NUM_NON_PAYLOAD_BYTES;
    }

private:
    MavlinkOrbSubscription *_sub;
    uint64_t _ca_traj_time;

    /* do not allow top copying this class */
    MavlinkStreamCaTrajectory(MavlinkStreamCaTrajectory &);
    MavlinkStreamCaTrajectory& operator = (const MavlinkStreamCaTrajectory &);

protected:
    explicit MavlinkStreamCaTrajectory(Mavlink *mavlink) : MavlinkStream(mavlink),
        _sub(_mavlink->add_orb_subscription(ORB_ID(ca_trajectory))), // make sure you
    enter the name of your uorb topic here
        _ca_traj_time(0)
    {}

    void send(const hrt_abstime t)
    {
        struct ca_traj_struct_s _ca_trajectory; //make sure ca_traj_struct_s is the
    definition of your uorb topic

        if (_sub->update(&_ca_traj_time, &_ca_trajectory)) {
            mavlink_ca_trajectory_t _msg_ca_trajectory; //make sure mavlink_ca_trajectory_t
    is the definition of your custom mavlink message

            _msg_ca_trajectory.timestamp = _ca_trajectory.timestamp;
            _msg_ca_trajectory.time_start_usec = _ca_trajectory.time_start_usec;
            _msg_ca_trajectory.time_stop_usec = _ca_trajectory.time_stop_usec;
            _msg_ca_trajectory.coefficients = _ca_trajectory.coefficients;
            _msg_ca_trajectory.seq_id = _ca_trajectory.seq_id;

            _mavlink->send_message(MAVLINK_MSG_ID_CA_TRAJECTORY, &_msg_ca_trajectory);
        }
    }
};


```

Finally append the stream class to the `streams_list` at the bottom of [mavlink_messages.cpp](#)

```

StreamListItem *streams_list[] = {
...
new StreamListItem(&MavlinkStreamCaTrajectory::new_instance, &MavlinkStreamCaTrajectory
y::get_name_static),
nullptr
};


```

Then make sure to enable the stream, for example by adding the following line to the startup script (`-r` configures the streaming rate, `-u` identifies the mavlink channel on UDP port 14556):

```
mavlink stream -r 50 -s CA_TRAJECTORY -u 14556
```

Receiving Custom MAVLink Messages

This section explains how to receive a message over mavlink and publish it to uORB.

Add a function that handles the incoming mavlink message in [mavlink_receiver.h](#)

```
#include <uORB/topics/ca_trajectory.h>
#include <v1.0/custom_messages/mavlink_msg_ca_trajectory.h>
```

Add a function that handles the incoming mavlink message in the `MavlinkReceiver` class in [mavlink_receiver.h](#)

```
void handle_message_ca_trajectory_msg(mavlink_message_t *msg);
```

Add an uORB publisher in the `MavlinkReceiver` class in [mavlink_receiver.h](#)

```
orb_advert_t _ca_traj_msg_pub;
```

Implement the `handle_message_ca_trajectory_msg` function in [mavlink_receiver.cpp](#)

```

void
MavlinkReceiver::handle_message_ca_trajectory_msg(mavlink_message_t *msg)
{
    mavlink_ca_trajectory_t traj;
    mavlink_msg_ca_trajectory_decode(msg, &traj);

    struct ca_traj_struct_s f;
    memset(&f, 0, sizeof(f));

    f.timestamp = hrt_absolute_time();
    f.seq_id = traj.seq_id;
    f.time_start_usec = traj.time_start_usec;
    f.time_stop_usec = traj.time_stop_usec;
    for(int i=0;i<28;i++)
        f.coefficients[i] = traj.coefficients[i];

    if (_ca_traj_msg_pub == nullptr) {
        _ca_traj_msg_pub = orb_advertise(ORB_ID(ca_trajectory), &f);

    } else {
        orb_publish(ORB_ID(ca_trajectory), _ca_traj_msg_pub, &f);
    }
}

```

and finally make sure it is called in [MavlinkReceiver::handle_message\(\)](#)

```

MavlinkReceiver::handle_message(mavlink_message_t *msg)
{
    switch (msg->msgid) {
        ...
    case MAVLINK_MSG_ID_CA_TRAJECTORY:
        handle_message_ca_trajectory_msg(msg);
        break;
        ...
    }
}

```

General

Set streaming rate

Sometimes it is useful to increase the streaming rate of individual topics (e.g. for inspection in QGC). This can be achieved by the following line

```
mavlink stream -u <port number> -s <mavlink topic name> -r <rate>
```

You can get the port number with `mavlink status` which will output (amongst others) transport protocol: UDP (<port number>) . An example would be

```
mavlink stream -u 14556 -s OPTICAL_FLOW_RAD -r 300
```

Daemons

A daemon is a process running in the background. In NuttX a daemon process is a task, in POSIX (Linux / Mac OS) a daemon is a thread.

New daemons are created through the `px4_task_spawn()` command.

```
daemon_task = px4_task_spawn_cmd("commander",
                                  SCHED_DEFAULT,
                                  SCHED_PRIORITY_DEFAULT + 40,
                                  3600,
                                  commander_thread_main,
                                  (char * const *)&argv[0]);
```

The arguments here are:

- arg0: the process name, `commander`
- arg1: the scheduling type (RR or FIFO)
- arg2: the scheduling priority
- arg3: the stack size of the new process or thread
- arg4: the task / thread main function
- arg5: a void pointer to pass to the new task, in this case holding the cmdline arguments.

Driver Development

The PX4 codebase uses a lightweight, unified driver abstraction layer: [DriverFramework](#). New drivers for POSIX and [QuRT](#) are written against this framework.

Legacy drivers for NuttX are based on the [Device] (<https://github.com/PX4/Firmware/tree/master/src/drivers/device>) framework and will be ported to DriverFramework.

Core Architecture

PX4 is a [reactive system](#) and uses pub/sub to transport messages. File handles are not required or used for the core operation of the system. Two main APIs are used:

- The publish / subscribe system which has a file, network or shared memory backend depending on the system PX4 runs on
- The global device registry, which allows to enumerate devices and get/set their configuration. This can be as simple as a linked list or map to the file system.

Bringing up a new Platform

NuttX

- The start script is located in [ROMFS/px4fmu_common](#)
- The OS configuration is located in [nuttx-configs](#). The OS gets loaded as part of the application build.
- The PX4 middleware configuration is located in [src/drivers/boards](#). It contains bus and GPIO mappings and the board initialization code.
- Drivers are located in [src/drivers](#)
- Reference config: Running 'make px4fmu-v4_default' builds the FMUv4 config, which is the current NuttX reference configuration

QuRT / Hexagon

- The start script is located in [posix-configs/](#)
- The OS configuration is part of the default Linux image (TODO: Provide location of LINUX IMAGE and flash instructions)
- The PX4 middleware configuration is located in [src/drivers/boards](#). TODO: ADD BUS

CONFIG

- Drivers are located in [DriverFramework](#)
- Reference config: Running 'make qurt_eagle_release' builds the Snapdragon Flight reference config

Device IDs

PX4 uses device IDs to identify individual sensors consistently across the system. These IDs are stored in the configuration parameters and used to match sensor calibration values, as well as to determine which sensor is logged to which logfile entry.

The order of sensors (e.g. if there is a `/dev/mag0` and an alternate `/dev/mag1`) is not determining priority - the priority is instead stored as part of the published uORB topic.

Decoding example

For the example of three magnetometers on a system, use the flight log (.px4log) to dump the parameters. The three parameters encode the sensor IDs and `MAG_PRIME` identifies which magnetometer is selected as the primary sensor. Each `MAGx_ID` is a 24bit number and should be padded left with zeros for manual decoding.

```
CAL_MAG0_ID = 73225.0
CAL_MAG1_ID = 66826.0
CAL_MAG2_ID = 263178.0
CAL_MAG_PRIME = 73225.0
```

This is the external HMC5983 connected via I2C, bus 1 at address `0x1E` : It will show up in the log file as `IMU.MagX` .

```
# device ID 73225 in 24-bit binary:
00000001 00011110 00001 001

# decodes to:
HMC5883 0x1E bus 1 I2C
```

This is the internal HMC5983 connected via SPI, bus 1, slave select slot 5. It will show up in the log file as `IMU1.MagX` .

```
# device ID 66826 in 24-bit binary:  
00000001 00000101 00001 010  
  
# decodes to:  
HMC5883 dev 5 bus 1 SPI
```

And this is the internal MPU9250 magnetometer connected via SPI, bus 1, slave select slot 4. It will show up in the log file as `IMU2.MagX`.

```
# device ID 263178 in 24-bit binary:  
00000100 00000100 00001 010  
  
#decodes to:  
MPU9250 dev 4 bus 1 SPI
```

Device ID Encoding

The device ID is a 24bit number according to this format. Note that the first fields are the least significant bits in the decoding example above.

```
struct DeviceStructure {  
    enum DeviceBusType bus_type : 3;  
    uint8_t bus: 5; // which instance of the bus type  
    uint8_t address; // address on the bus (eg. I2C address)  
    uint8_t devtype; // device class specific device type  
};
```

The `bus_type` is decoded according to:

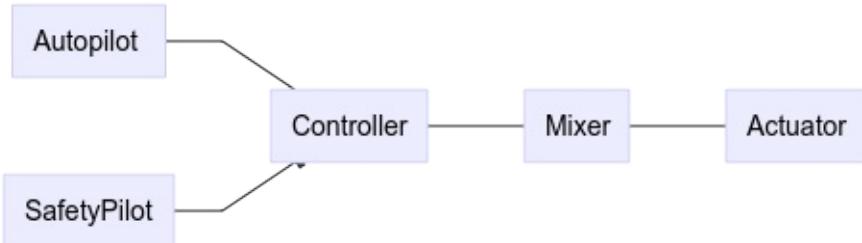
```
enum DeviceBusType {  
    DeviceBusType_UNKNOWN = 0,  
    DeviceBusType_I2C = 1,  
    DeviceBusType_SPI = 2,  
    DeviceBusType_UAVCAN = 3,  
};
```

and `devtype` is decoded according to:

```
#define DRV_MAG_DEVTYPE_HMC5883 0x01
#define DRV_MAG_DEVTYPE_LSM303D 0x02
#define DRV_MAG_DEVTYPE_ACCELSIM 0x03
#define DRV_MAG_DEVTYPE_MPU9250 0x04
#define DRV_ACC_DEVTYPE_LSM303D 0x11
#define DRV_ACC_DEVTYPE_BMA180 0x12
#define DRV_ACC_DEVTYPE_MPU6000 0x13
#define DRV_ACC_DEVTYPE_ACCELSIM 0x14
#define DRV_ACC_DEVTYPE_GYROSIM 0x15
#define DRV_ACC_DEVTYPE_MPU9250 0x16
#define DRV_GYR_DEVTYPE_MPU6000 0x21
#define DRV_GYR_DEVTYPE_L3GD20 0x22
#define DRV_GYR_DEVTYPE_GYROSIM 0x23
#define DRV_GYR_DEVTYPE_MPU9250 0x24
#define DRV_RNG_DEVTYPE_MB12XX 0x31
#define DRV_RNG_DEVTYPE_LL40LS 0x32
```

Airframe Overview

The PX4 system is architected in a modular fashion allowing it to use a single codebase for all robot types.



Basic Equipment

All hardware configurations in the airframe section assume a basic set of equipment:

- A Taranis Plus remote control for the safety pilot (or equivalent with PPM / S.BUS out)
- A ground control station
 - Samsung Note 4 or equivalent / more recent Android tablet
 - iPad (requires Wifi telemetry adapter)
 - Any MacBook or Ubuntu Linux laptop
- An in-field computer (for software developers)
 - MacBook Pro or Air with OS X 10.10 or newer
 - Modern laptop with Ubuntu Linux (14.04 or newer)
- Safety glasses
- For multicopters: A tether for more risky tests

PX4 can be used with a much wider range of equipment, but new developers will benefit from going with one of the standard setups, and a Taranis RC plus a Note 4 tablet make up for a very inexpensive field kit.

Adding a new Airframe Configuration

PX4 uses canned configurations as starting point for airframes. Adding a configuration is straightforward: Create a new file which is prepended with a free autostart ID in the [init.d folder](#) and [build and upload](#) the software.

Developers not wanting to create their own configuration can instead customize existing configurations using textfiles on the microSD card, as detailed on the [custom system startup](#) page.

Airframe configurations

An airframe configuration consists of three main blocks:

- The apps it should start, e.g. multicopter or fixed wing controllers
- The physical configuration of the system (e.g. a plane, wing or multicopter). This is called mixer.
- Tuning gains

These three aspects are mostly independent, which means that many configurations share the same physical layout of the airframe and start the same applications and most differ in their tuning gains.

All configurations are stored in the [ROMFS/px4fmu_common/init.d](#) folder. All mixers are stored in the [ROMFS/px4fmu_common/mixers](#) folder.

Config file

A typical configuration file is below.

```
#!nsh
#
# @name Wing Wing (aka Z-84) Flying Wing
#
# @url https://pixhawk.org/platforms/planes/z-84_wing_wing
#
# @type Flying Wing
#
# @output MAIN1 left aileron
# @output MAIN2 right aileron
# @output MAIN4 throttle
#
# @output AUX1 feed-through of RC AUX1 channel
# @output AUX2 feed-through of RC AUX2 channel
# @output AUX3 feed-through of RC AUX3 channel
#
# @maintainer Lorenz Meier <lorenz@px4.io>
#
sh /etc/init.d/rc.fw_defaults

if [ $AUTOCNF == yes ]
then
    param set BAT_N_CELLS 2
    param set FW_AIRSPD_MAX 15
    param set FW_AIRSPD_MIN 10
    param set FW_AIRSPD_TRIM 13
    param set FW_ATT_TC 0.3
    param set FW_L1_DAMPING 0.74
    param set FW_L1_PERIOD 16
    param set FW_LND_ANG 15
    param set FW_LND_FLALT 5
    param set FW_LND_HHDIST 15
    param set FW_LND_HVIRT 13
    param set FW_LND_TLALT 5
    param set FW_THR_LND_MAX 0
    param set FW_PR_FF 0.35
    param set FW_RR_FF 0.6
    param set FW_RR_P 0.04
fi

# Configure this as plane
set MAV_TYPE 1
# Set mixer
set MIXER wingwing
# Provide ESC a constant 1000 us pulse
set PWM_OUT 4
set PWM_DISARMED 1000
```

IMPORTANT REMARK: If you want to reverse a channel, never do this neither on your RC transmitter nor with e.g. `RC1_REV`. The channels are only reversed when flying in manual mode, when you switch in an autopilot flight mode, the channels output will still be wrong (it only inverts your RC signal). Thus for a correct channal assignment change either your PWM signals with `PWM_MAIN_REV1` (e.g. for channel one) or change the signs for both output scaling and output range in the corresponding mixer (see below).

Mixer file

A typical configuration file is below.

The plugs of the servos / motors go in the order of the mixers in this file. So MAIN1 would be the left aileron, MAIN2 the right aileron, MAIN3 is empty (note the Z: zero mixer) and MAIN4 is throttle (to keep throttle on output 4 for common fixed wing configurations).

A mixer is encoded in normalized units from -10000 to 10000, corresponding to -1..+1.

```

M: 2
O:      10000  10000      0 -10000  10000
S: 0 0  -6000  -6000      0 -10000  10000
S: 0 1   6500   6500      0 -10000  10000

```

Where each number from left to right means:

- M: Indicates two scalers for two inputs
- O: Indicates the output scaling (*1 in negative*, 1 in positive), offset (zero here), and output range (-1..+1 here). If you want to invert your PWM signal, the signs for both output scalings and both output range numbers have to be changed. (`o: -10000 -10000 0 10000 -10000`)
- S: Indicates the first input scaler: It takes input from control group #0 (attitude controls) and the first input (roll). It scales the input * 0.6 and reverts the sign (-0.6 becomes -6000 in scaled units). It applies no offset (0) and outputs to the full range (-1..+1)
- S: Indicates the second input scaler: It takes input from control group #0 (attitude controls) and the second input (pitch). It scales the input * 0.65 and reverts the sign (-0.65 becomes -6500 in scaled units). It applies no offset (0) and outputs to the full range (-1..+1)

Both scalers are added, which for a flying wing means the control surface takes maximum 60% deflection from roll and 65% deflection from pitch. As it is over-committed with 125% total deflection for maximum pitch and roll, it means the first channel (roll here) has priority over the second channel / scaler (pitch).

The complete mixer looks like this:

```
Delta-wing mixer for PX4FMU
=====
```

```
Designed for Wing Wing Z-84
```

This file defines mixers suitable for controlling a delta wing aircraft using PX4FMU. The configuration assumes the elevon servos are connected to PX4FMU servo outputs 0 and 1 and the motor speed control to output 3. Output 2 is assumed to be unused.

Inputs to the mixer come from channel group 0 (vehicle attitude), channels 0 (roll), 1 (pitch) and 3 (thrust).

See the README for more information on the scaler format.

```
Elevon mixers
```

```
-----
```

```
Three scalers total (output, roll, pitch).
```

On the assumption that the two elevon servos are physically reversed, the pitch input is inverted between the two servos.

The scaling factor for roll inputs is adjusted to implement differential travel for the elevons.

```
M: 2
```

```
O:      10000  10000      0 -10000  10000
S: 0 0  -6000  -6000      0 -10000  10000
S: 0 1   6500   6500      0 -10000  10000
```

```
M: 2
```

```
O:      10000  10000      0 -10000  10000
S: 0 0  -6000  -6000      0 -10000  10000
S: 0 1  -6500  -6500      0 -10000  10000
```

```
Output 2
```

```
-----
```

```
This mixer is empty.
```

```
Z:
```

```
Motor speed mixer
```

```
-----
```

```
Two scalers total (output, thrust).
```

This mixer generates a full-range output (-1 to 1) from an input in the (0 - 1) range. Inputs below zero are treated as zero.

```
M: 1
```

```
O:      10000  10000      0 -10000  10000
S: 0 3     0  20000 -10000 -10000  10000
```


Multicopter Airframes



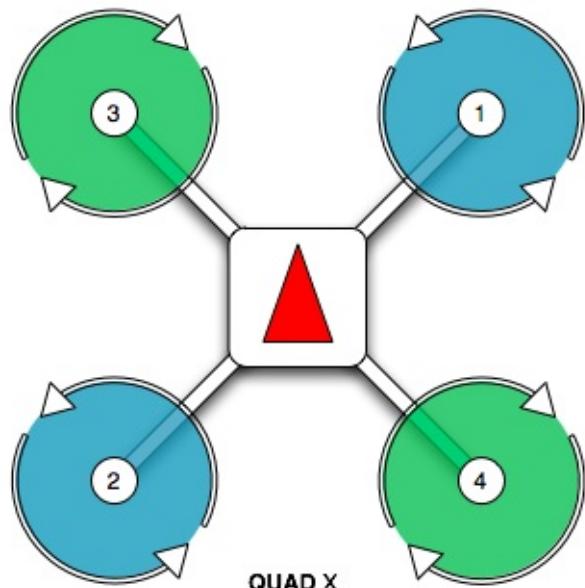
[Video link](#)

Airframe Motor Map

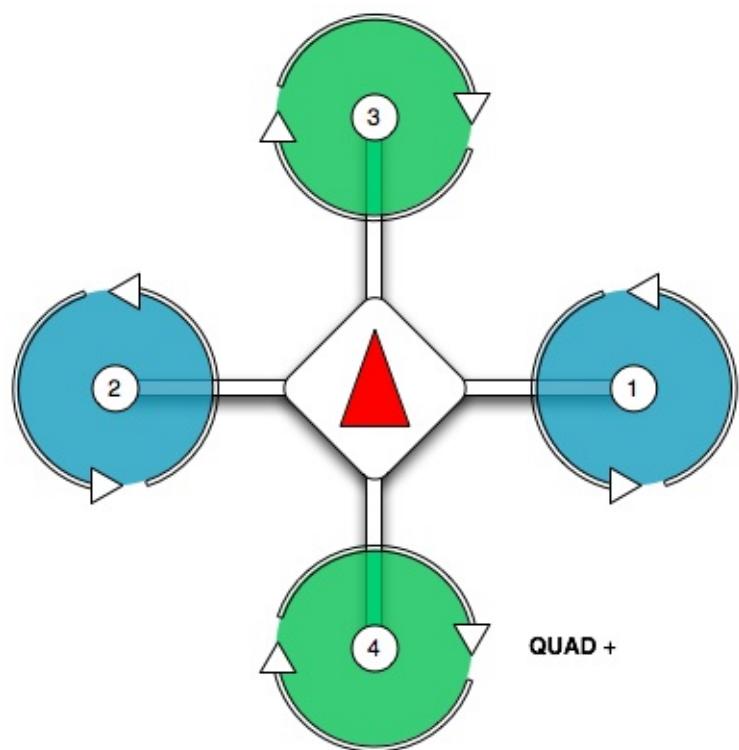
The motor maps below indicate which `MAIN` output connects to which motor. It also indicates the turn direction of the propeller.

The turn direction of a 3-phase motor can be reversed by flipping any two of the three connected wires.

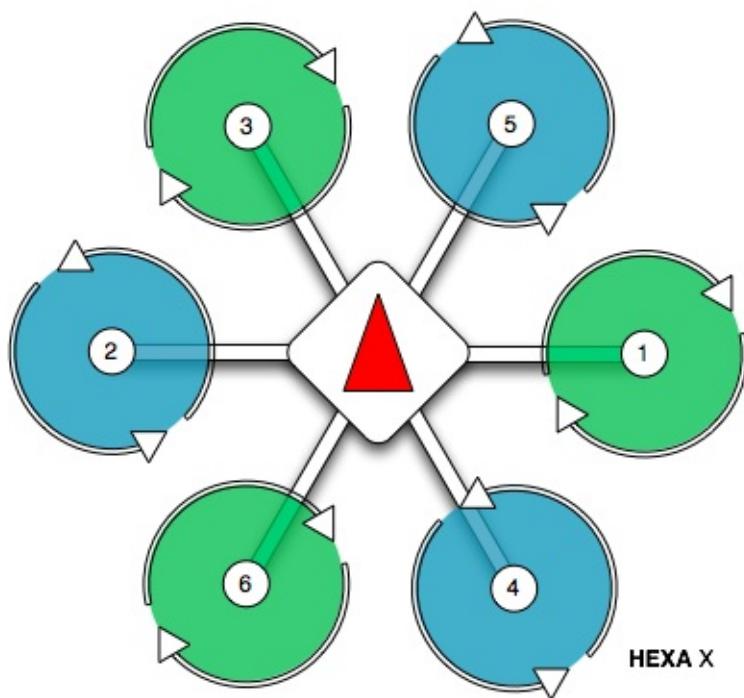
Quadrotor X Layout



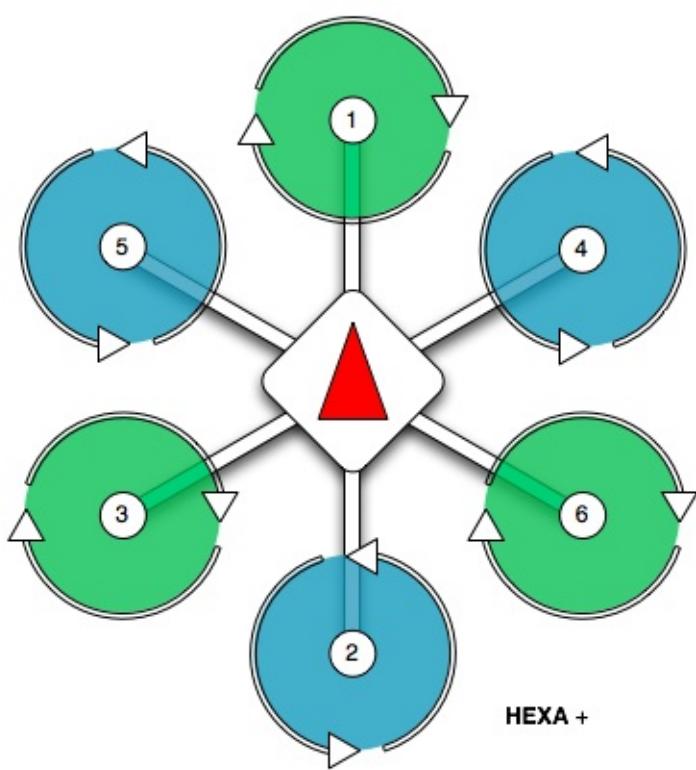
Quadrotor Plus Layout



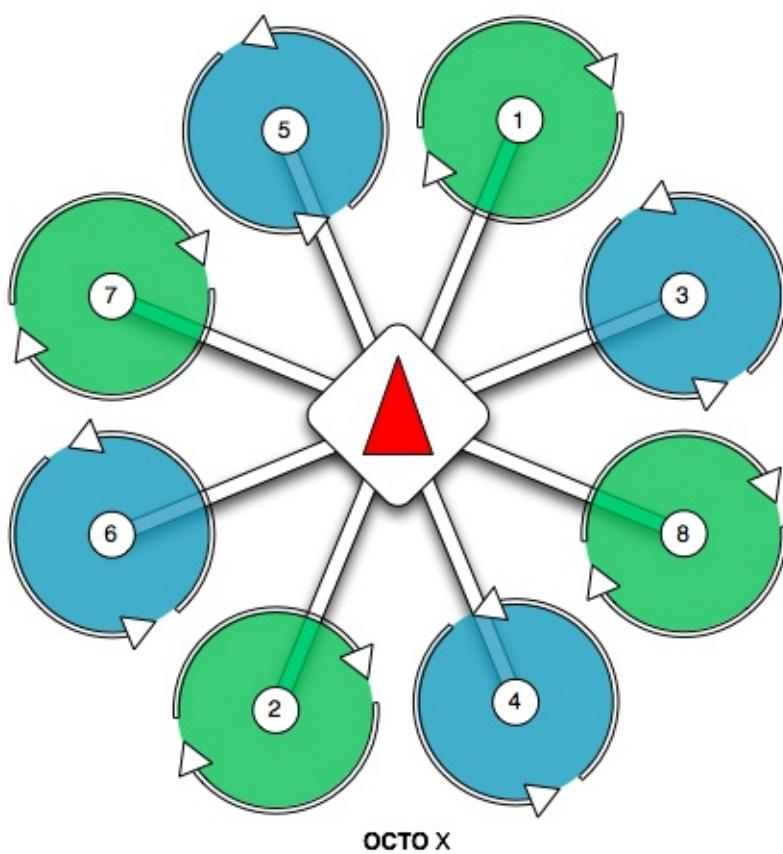
Hexarotor X Layout



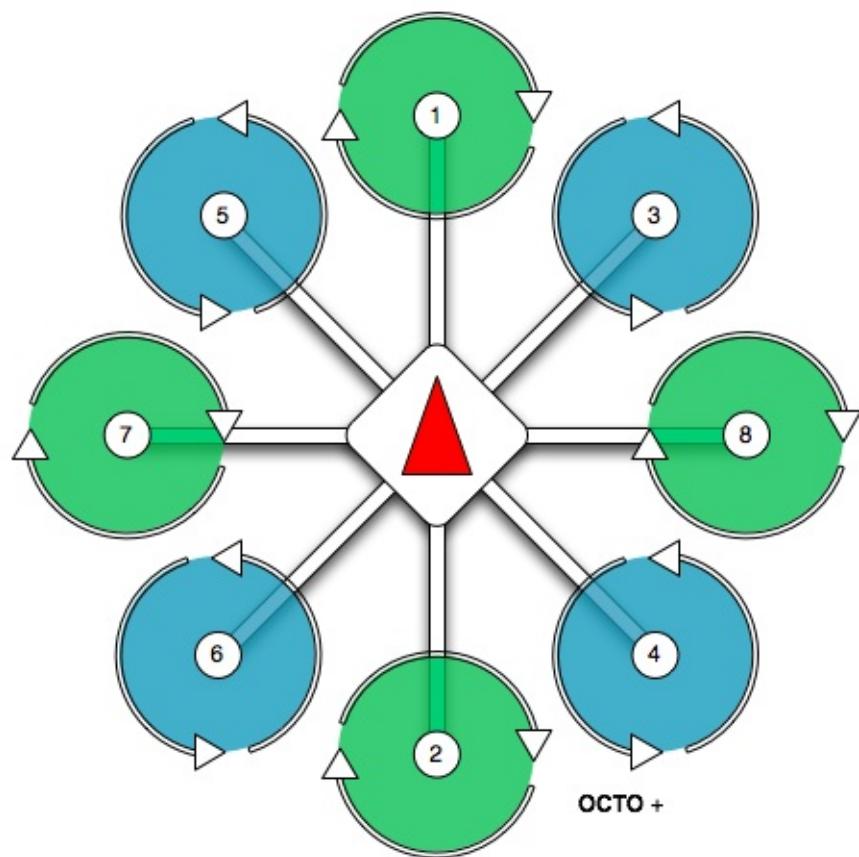
Hexarotor Plus Layout



Octorotor X Layout



Octorotor Plus Layout



Tricopters and coaxial hexacopters are supported as well, but we have no graphics yet.

QAV 250

Parts List

- [Pixracer kit](#) or [Pixfalcon kit](#) (including GPS and power module)
- [Mini telemetry set](#) for HKPilot32

Motor Connections

Output	Rate	Actuator
MAIN1	400 Hz	Front right, CCW
MAIN2	400 Hz	Back left, CCW
MAIN3	400 Hz	Front left, CW
MAIN4	400 Hz	Back right, CW
AUX1	50 Hz	RC AUX1
AUX2	50 Hz	RC AUX2
AUX3	50 Hz	RC AUX3

Matrice 100



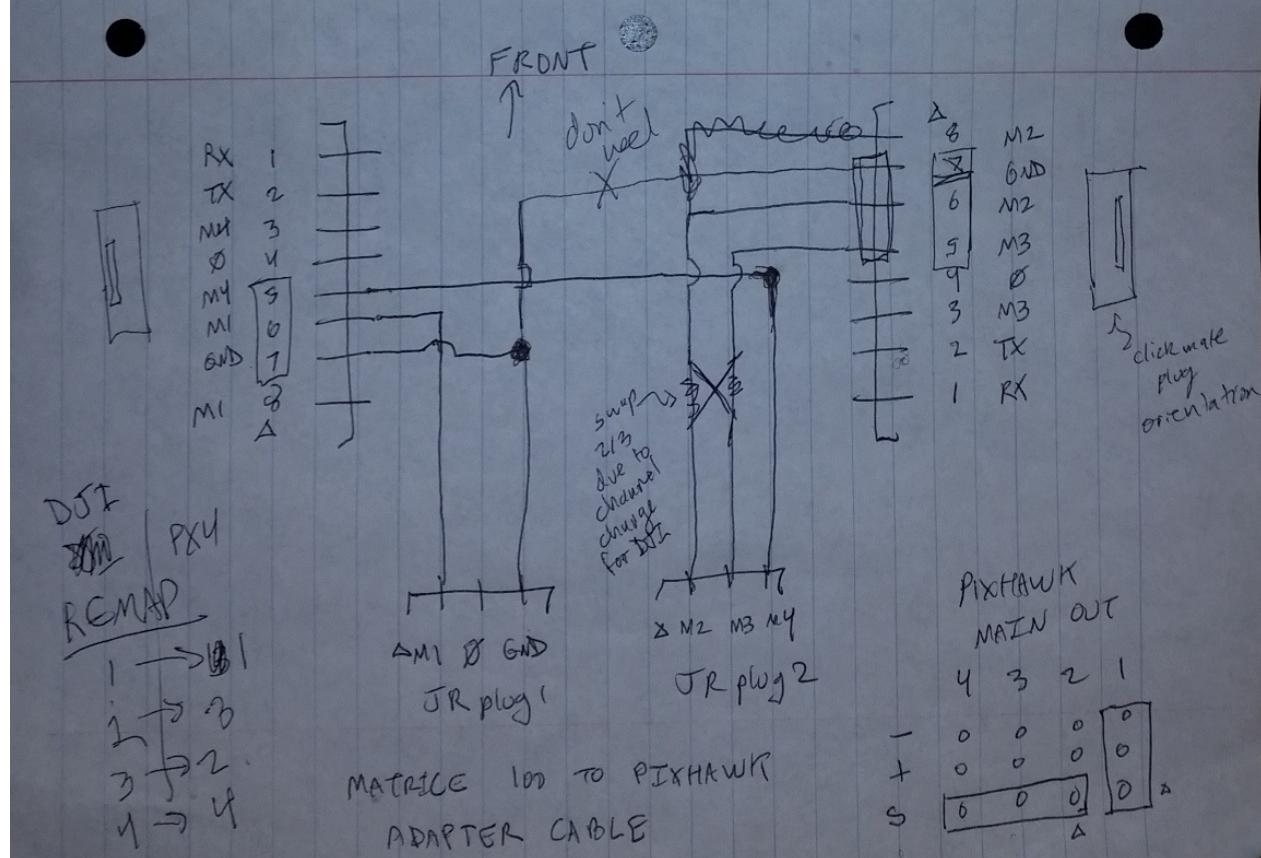
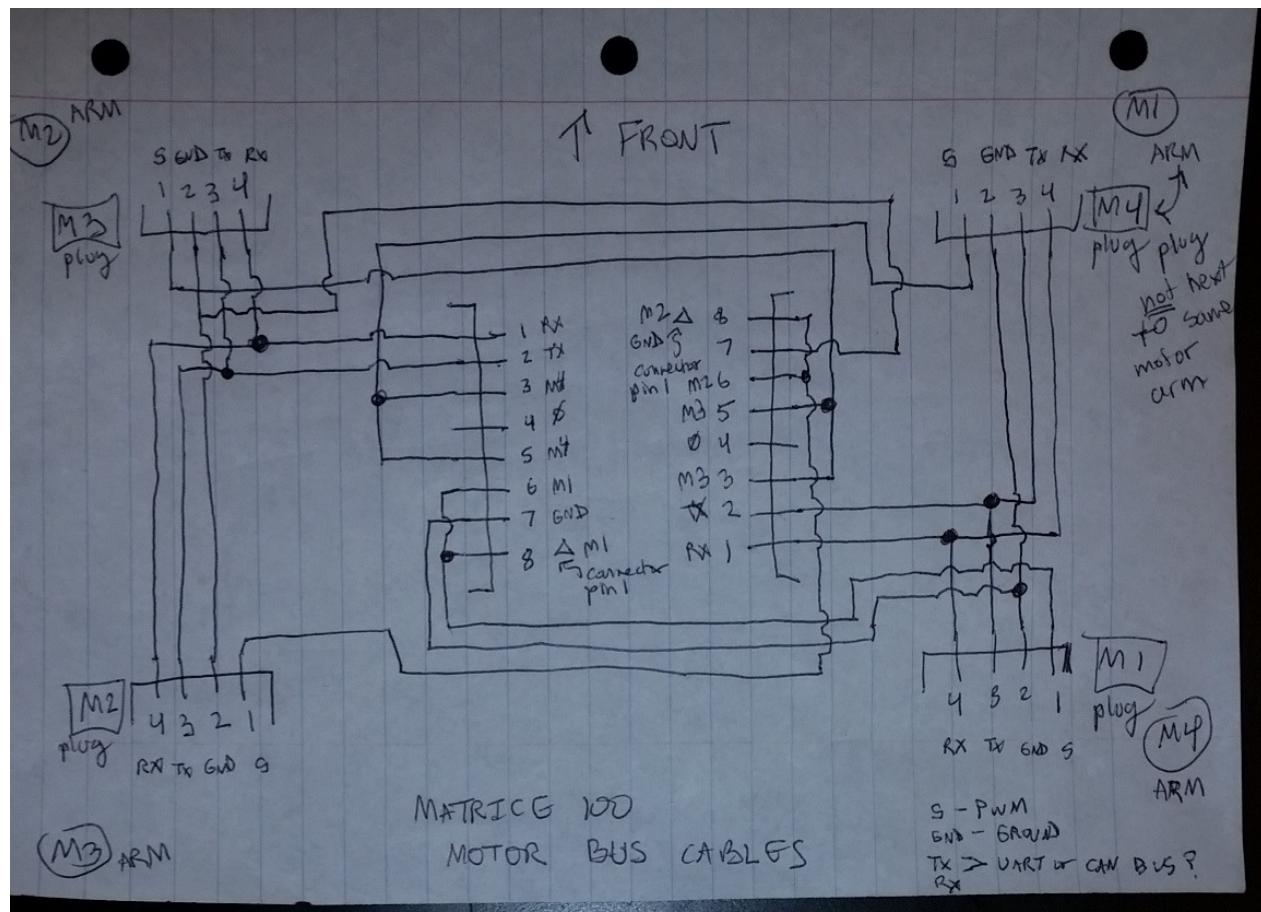
[Video link](#)

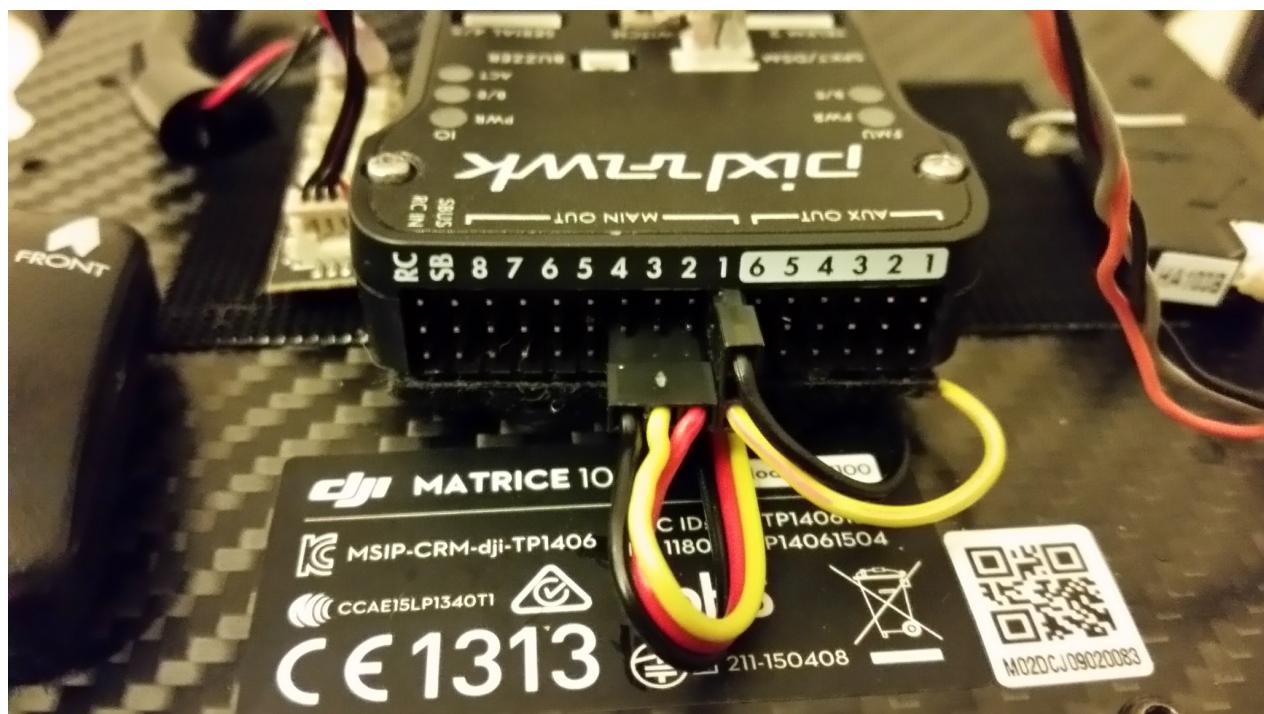
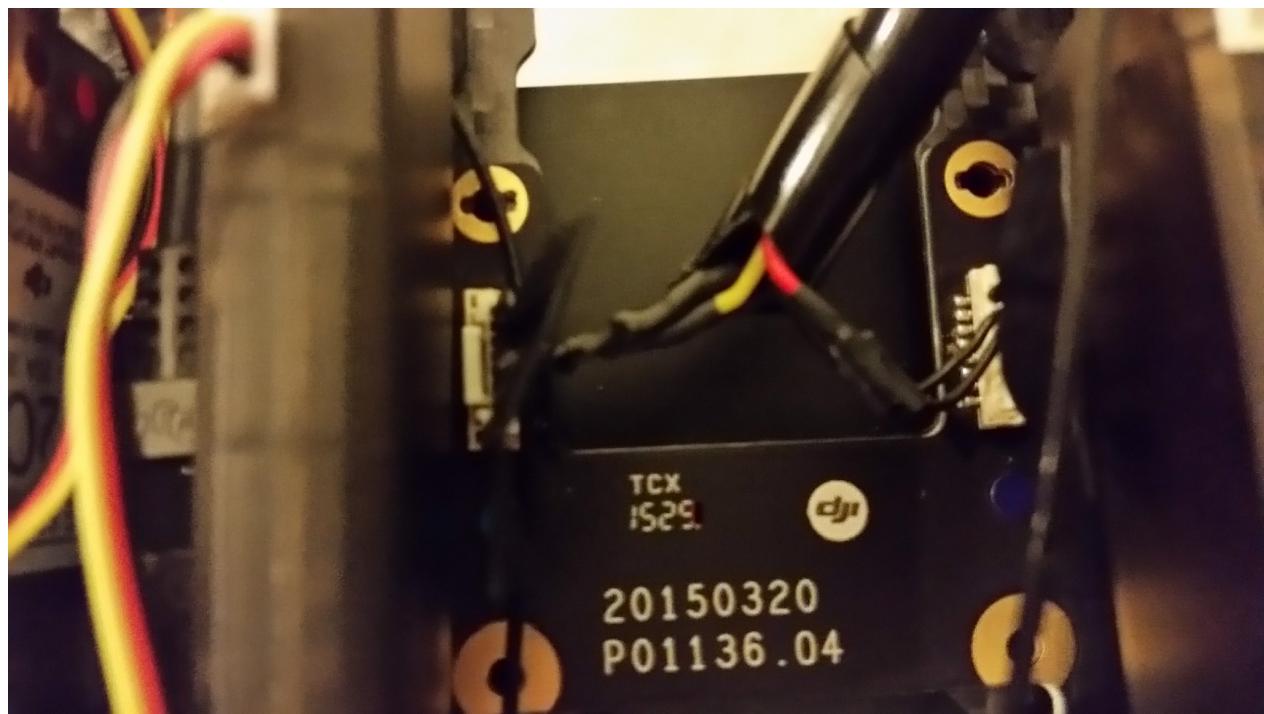


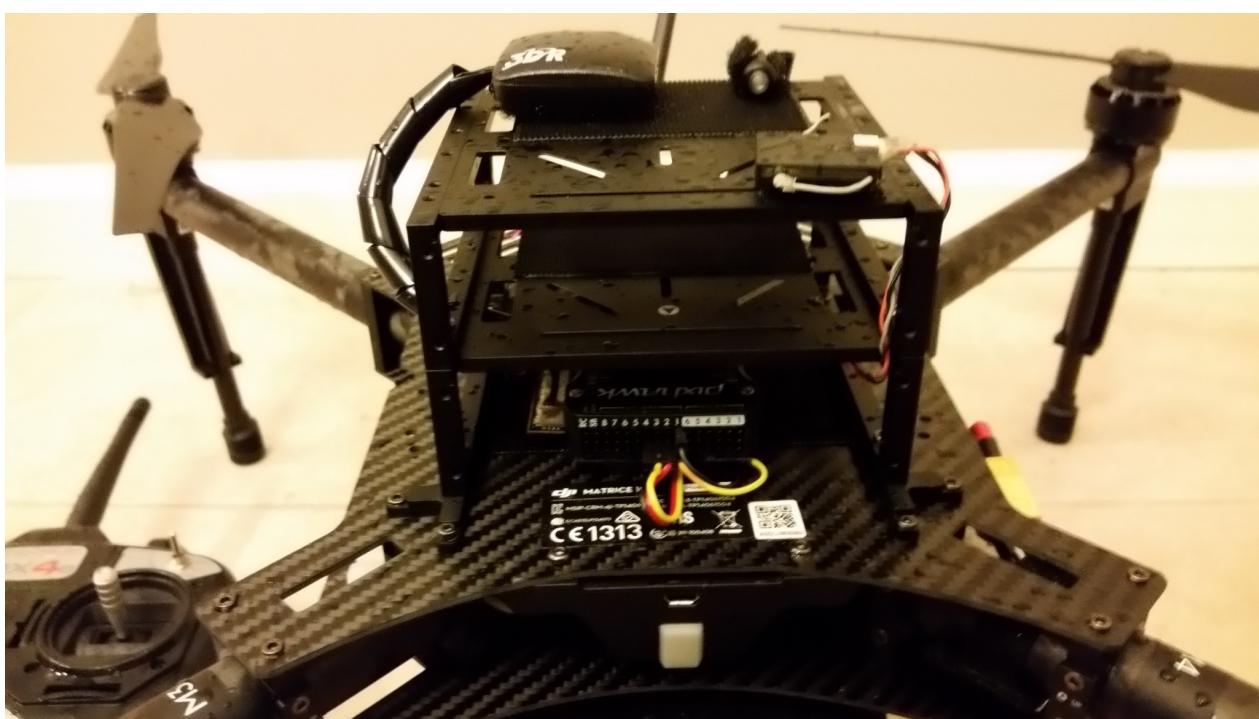
Parts List

- [DJI Matrice 100 Just ESC's motors, and frame.](#)

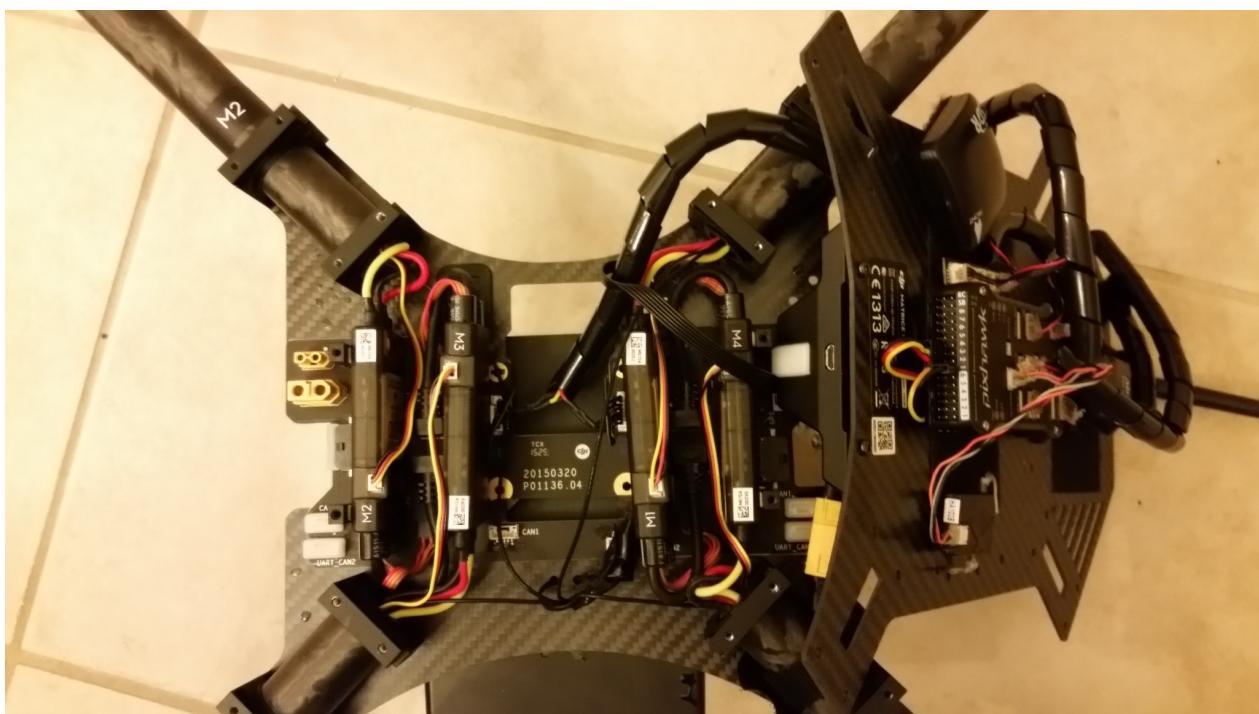
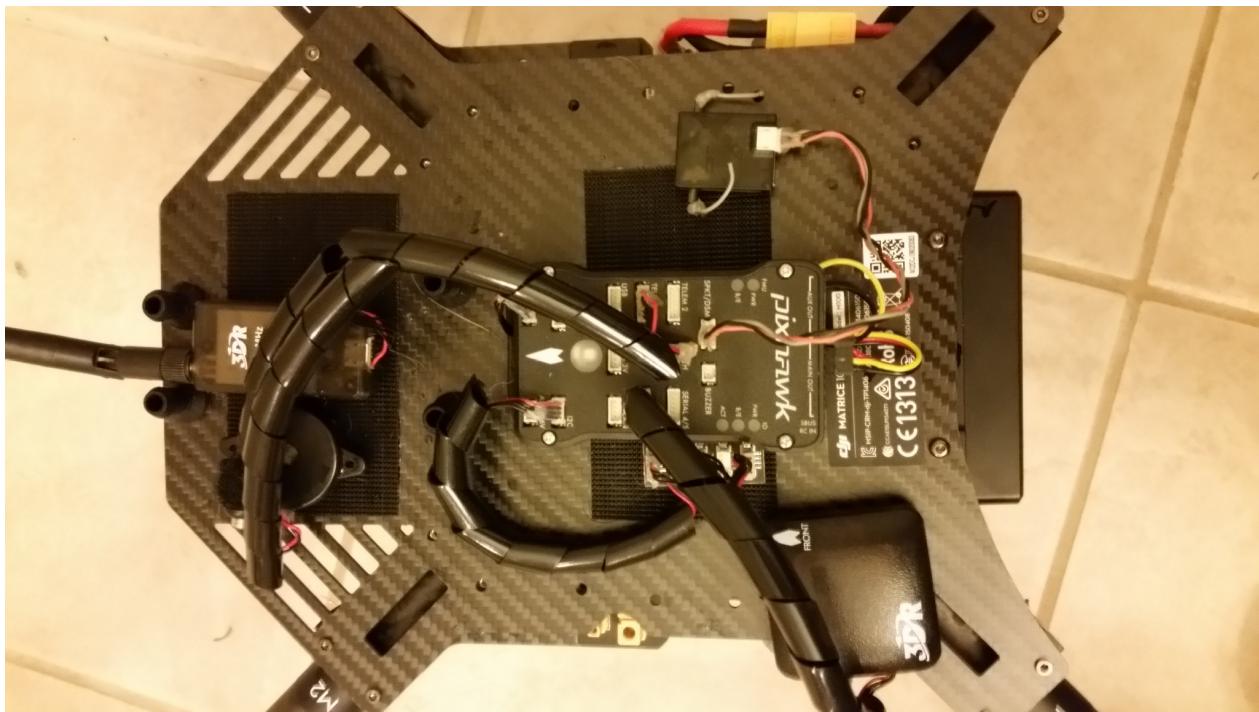
Motor Connections







Matrice 100



Output	Rate	Actuator
MAIN1	400 Hz	Front right, CCW
MAIN2	400 Hz	Back left, CCW
MAIN3	400 Hz	Front left, CW
MAIN4	400 Hz	Back right, CW
AUX1	50 Hz	RC AUX1
AUX2	50 Hz	RC AUX2
AUX3	50 Hz	RC AUX3

Parameters

- At high throttle the inner loop causes oscillations with default x quad gains. At low throttle, higher gains give a better response, this suggests that some gain scheduling based on the throttle may improve the overall response and this could be implemented in mc_att_control. For now we will just tune it so that there are no oscillations at low or high throttle, and take the bandwidth hit at low throttle.
 - MC_PITCHRATE_P: 0.05
 - MC_PITCHRATE_D: 0.001
- The battery has 6 cells instead of the default 3
 - BAT_N_CELLS: 6

QAV-R



[Video link](#)

Parts List

Frame/ Motors

- QAV-R FPV Racing Quadcopter (5")
- Lumenier 4Power Quick Swap Power Distribution Board
- Lumenier RX2204-14 2300Kv Motor
- Lumenier Mini 20A ESC w/ SimonK AutoShot, 5V/1A BEC (2-4s)
- Gemfan 5x4.5 Nylon Glass Fiber Propeller (Set of 4 - Black)
- Lumenier 1300mAh 3s 60c Lipo Battery (XT60)

FPV Gear

- QAV180/210 Carbon Fiber Vibration Damping Camera Plate (GoPro+Mobius)
- Fat Shark 900TVL CCD FPV Camera (NTSC)
- FatShark 5.8ghz 250mW A/V Transmitter

Accessories

- Professional Travel Case for the QAV-R (nice to have)

Plane Airframes

PX4 supports every imaginable plane geometry through its flexible [mixing system](#), including normal planes, flying wings, inverted V-tail planes vertical takeoff and landing planes.



[Video link](#)

Wing Wing Z-84 Airframe

Parts List

Ensure to order the PNF version which includes motor, propeller and electronic speed controller. The kit version requires these to be purchased individually.

- Zeta Science Wing-Wing Z-84 PNF ([Hobbyking Store](#))
- 1800 mAh 2S LiPo
 - [Team Orion 1800mAh 7.4V 50C 2S1P](#)
- FrSky D4R-II receiver or equivalent (jumpered to PPM sum output according to its manual)
- [Pixracer kit](#) (including GPS and power module)
- [Mini telemetry set](#) for HKPilot32
- [Digital airspeed sensor](#) for HKPilot32 / Pixfalcon
- Spare parts
 - [O-Rings for prop saver](#)
 - [Replacement propellers](#)

Servo Connections

Output	Rate	Actuator
MAIN1	50 Hz	Left aileron servo
MAIN2	50 Hz	Right aileron servo
MAIN3	50 Hz	Empty
MAIN4	50 Hz	Motor controller

VTOL Airframes

The [PX4 Flight Stack](#) supports virtually all VTOL configurations:

- Tailsitters (duo and quadrotors in X and plus configuration)
- Tiltrotors (Firefly Y6)
- Standard plane VTOL (plane plus quad)

The VTOL codebase is the same codebase as for all other airframes and just adds additional control logic, in particular for transitions.

All these VTOL configurations have been actively test-flown and are ready to use. Ensure to have an airspeed sensor attached to the system as its used by the autopilot when its safe to perform the transition.

Key Configuration Parameters

These configuration parameters have to be set correctly when creating a new airframe configuration.

- `VT_FW_PERM_STAB` the system always uses attitude stabilization in hover mode. If this parameter is set to 1, the plane mode also defaults to attitude stabilization. If it is set to 0, it defaults to pure manual flight.
- `VT_ARSP_TRANS` is the airspeed in m/s at which the plane transitions into forward flight. Setting it too low can cause a stall during the transition.
- `RC_MAP_TRANS_SW` should be assigned to a RC switch before flight. This allows you to check if the multicopter- and fixed wing mode work properly. (Can also be used to switch manually between those two control modes during flight)

Tailsitter

The [build log](#) contains further detail.



[Video link](#)

Tiltrotor

The [build log](#) contains all settings and instructions to get one of these up and running.



[Video link](#)

Standard Plane VTOL

The [build log](#) contains further instructions how to build and reproduce the results below.



[Video link](#)



[Video link](#)

VTOL Testing

How-to test that the VTOL functions properly, main focus are transitions:

- On the bench
- In flight

General notes on transitions

There are currently 3 ways of commanding the VTOL to transition:

- RC switch (2 pos, aux1)
- MAVLink command (MAV_CMD_DO_VTOL_TRANSITION)
- Transition during mission (MAV_CMD_DO_VTOL_TRANSITION internally)

When a transition is commanded (by either of the methods above), the VTOL enters the transition phase. If the VTOL receives a new transition command back to the old state during an ongoing transition it will switch back instantly. This is a safety feature to abort the transition when necessary. After the transition has been completed, the VTOL will be in the new state and a commanded transition into the reverse direction will take place normally.

Make sure the AUX1 channel is assigned to an RC switch and that airspeed is working properly.

On the bench

Remove all props! To test transition functionality properly, the vehicle needs to be armed.

By default, starting in multirotor mode:

- arm the vehicle
- check that motors are running in multirotor configuration (rudders/elevons should not move on roll/pitch/yaw inputs)
- toggle transition switch
- (if applicable) wait on step 1 of the transition phase to complete
- blow into pitot tube to simulate airspeed
- (if applicable) step 2 of the transition phase will be executed
- check that motors are running in fixed-wing configuration (roll/pitch/yaw inputs should control rudders/elevons)
- toggle transition switch

- observe back transition
- check that motors are running in multirotor configuration (rudders/elevons should not move on roll/pitch/yaw inputs)

In flight

Before testing transitions in flight, make sure the VTOL flies stable in multirotor mode. In general, if something doesn't go as planned, transition to multirotor mode and let it recover (it does a good job when it's properly tuned).

In-flight transition requires at least the following parameters to match your airframe and piloting skills:

Param	Notes
VT_FW_PERM_STAB	Turns permanent stabilization on/off for fixed-wing.
VT_ARSP_BLEND	At which airspeed the fixed-wing controls become active.
VT_ARSP_TRANS	At which airspeed the transition to fixed-wing is complete.

There are more parameters depending on the type of VTOL, see the [parameter reference](#).

Manual transition test

The basic procedure to test manual transitions is as follows:

- arm and takeoff in multirotor mode
- climb to a safe height to allow for some drop after transition
- turn into the wind
- toggle transition switch
- observe transition (**MC-FW**)
- fly in fixed-wing
- come in at a safe height to allow for some drop after transition
- toggle transition switch
- observe transition (**FW-MC**)
- land and disarm

MC-FW

During the transition from MC to FW the following can happen:

1. it loses control while gaining speed (this can happen due to many factors)
2. the transition takes too long and it flies too far away before the transition finishes

For 1): Switch back to multirotor (will happen instantly). Try to identify the problem (check setpoints).

For 2): If blending airspeed is set and it has a higher airspeed already it is controllable as fixed-wing. Therefore it is possible to fly around and give it more time to finish the transition. Otherwise switch back to multirotor and try to identify the problem (check airspeed).

FW-MC

The transition from FW to MC is mostly unproblematic. In-case it seems to loose control the best approach is to let it recover.

Automatic transition test (mission, commanded)

Commanded transitions only work in auto (mission) or offboard flight-mode. Make sure you are confident to operate the auto/offboard and transition switch in flight.

Switching to manual will reactivate the transition switch. For example: if you switch out of auto/offboard when in automatic fixed-wing flight and the transition switch is currently in multirotor position it will transition to multirotor right away.

Procedure

The following procedure can be used to test a mission with transition:

- upload mission
- takeoff in multirotor mode and climb to mission height
- enable mission with switch
- observe transition to fixed-wing flight
- enjoy flight
- observe transition back to multirotor mode
- disable mission
- land manually

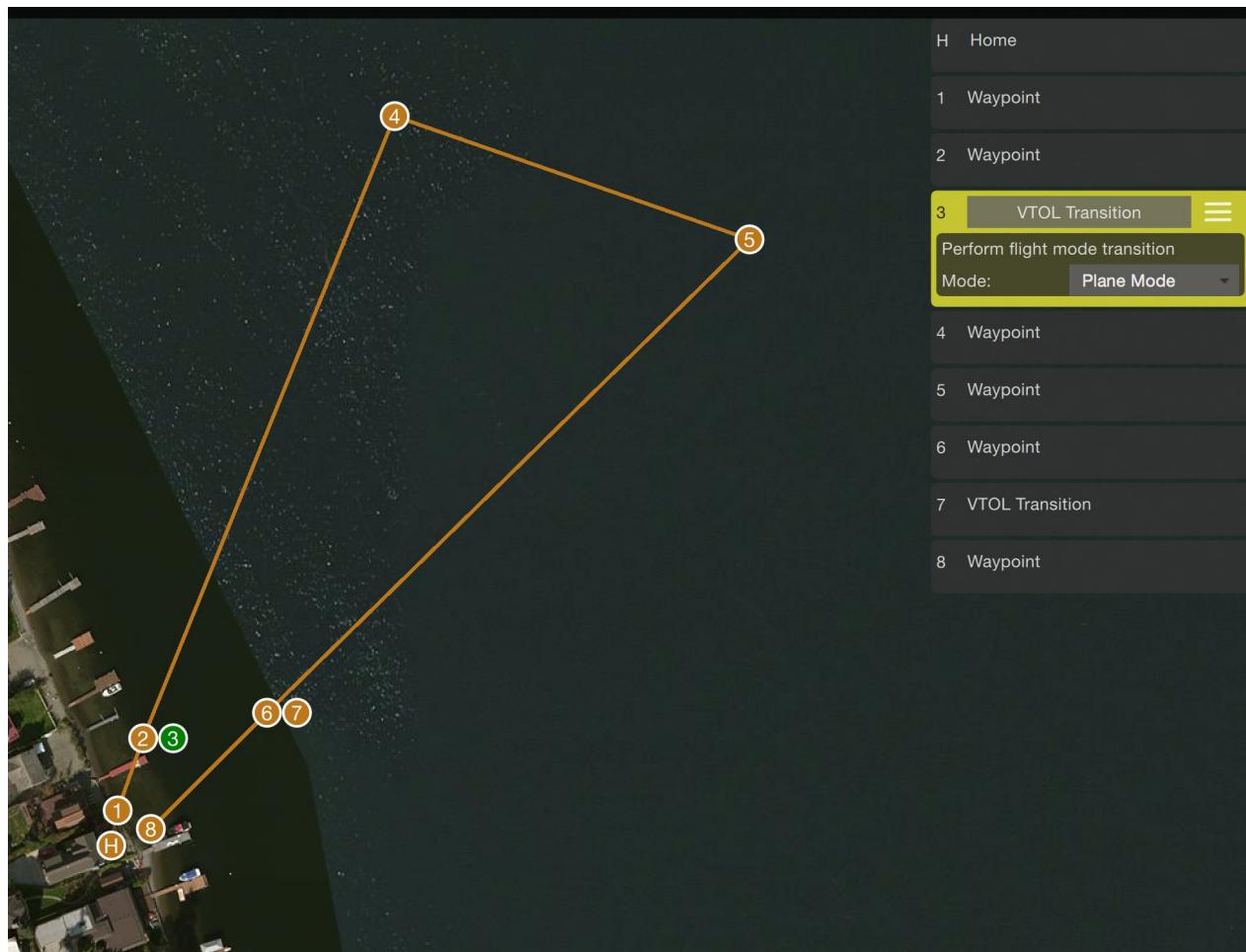
During flight, the manual transition switch stays in multirotor position. If something doesn't go as planned, switch to manual and it will recover in multirotor mode.

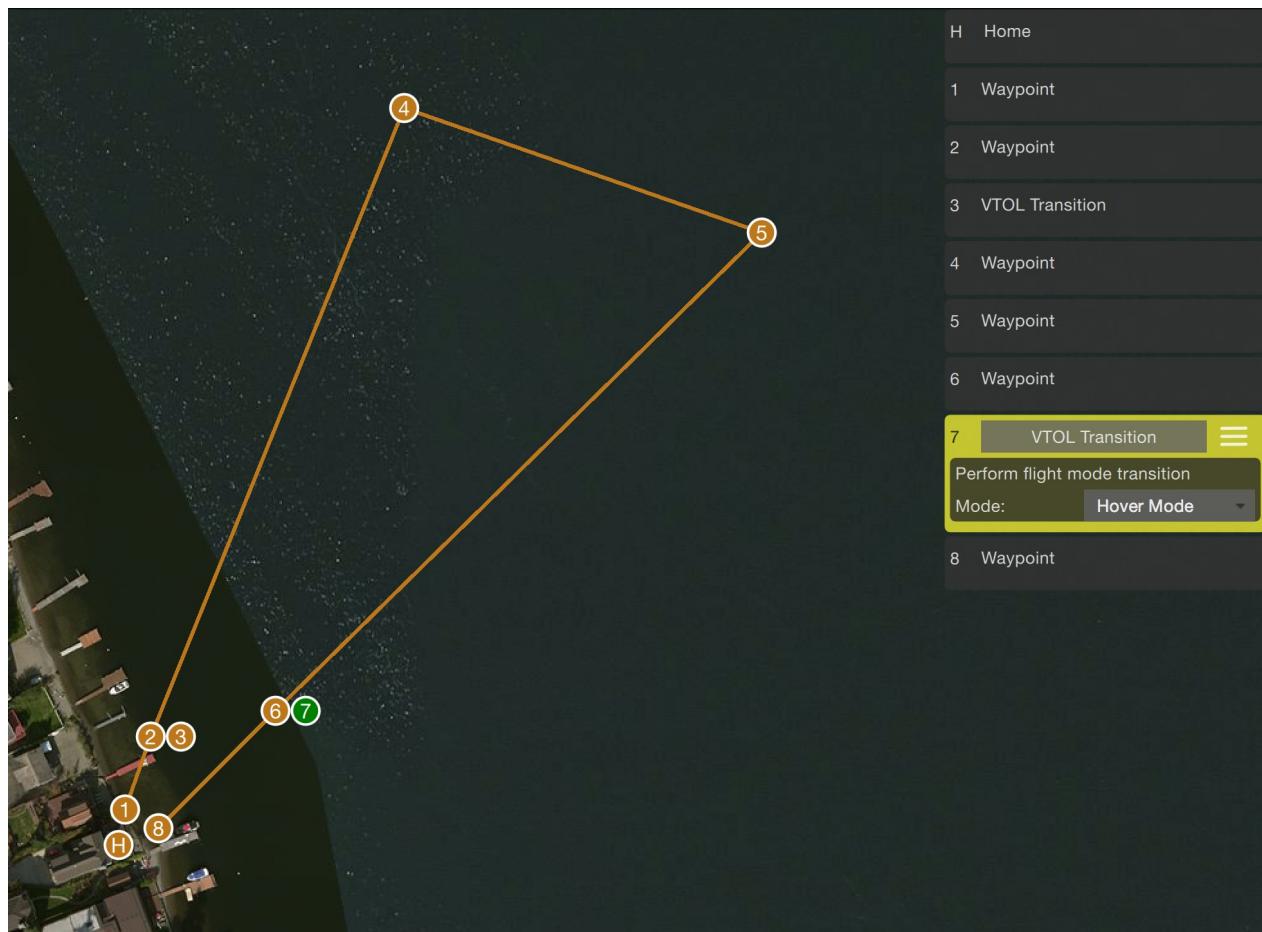
Example mission

The mission should contain at least (also see screenshots below):

- (1) position waypoint near takeoff location
- (2) position waypoint in the direction of the planned fixed-wing flight route
- (3) transition waypoint (to plane mode)

- (4) position waypoint further away (at least as far away as the transition needs)
- (6) position waypoint to fly back (a bit before takeoff location so back transition takes some distance)
- (7) transition waypoint (to hover mode)
- (8) position waypoint near takeoff location





TBS Caipiroshka

The Caipiroshka VTOL is a slightly modified TBS Caipirinha.



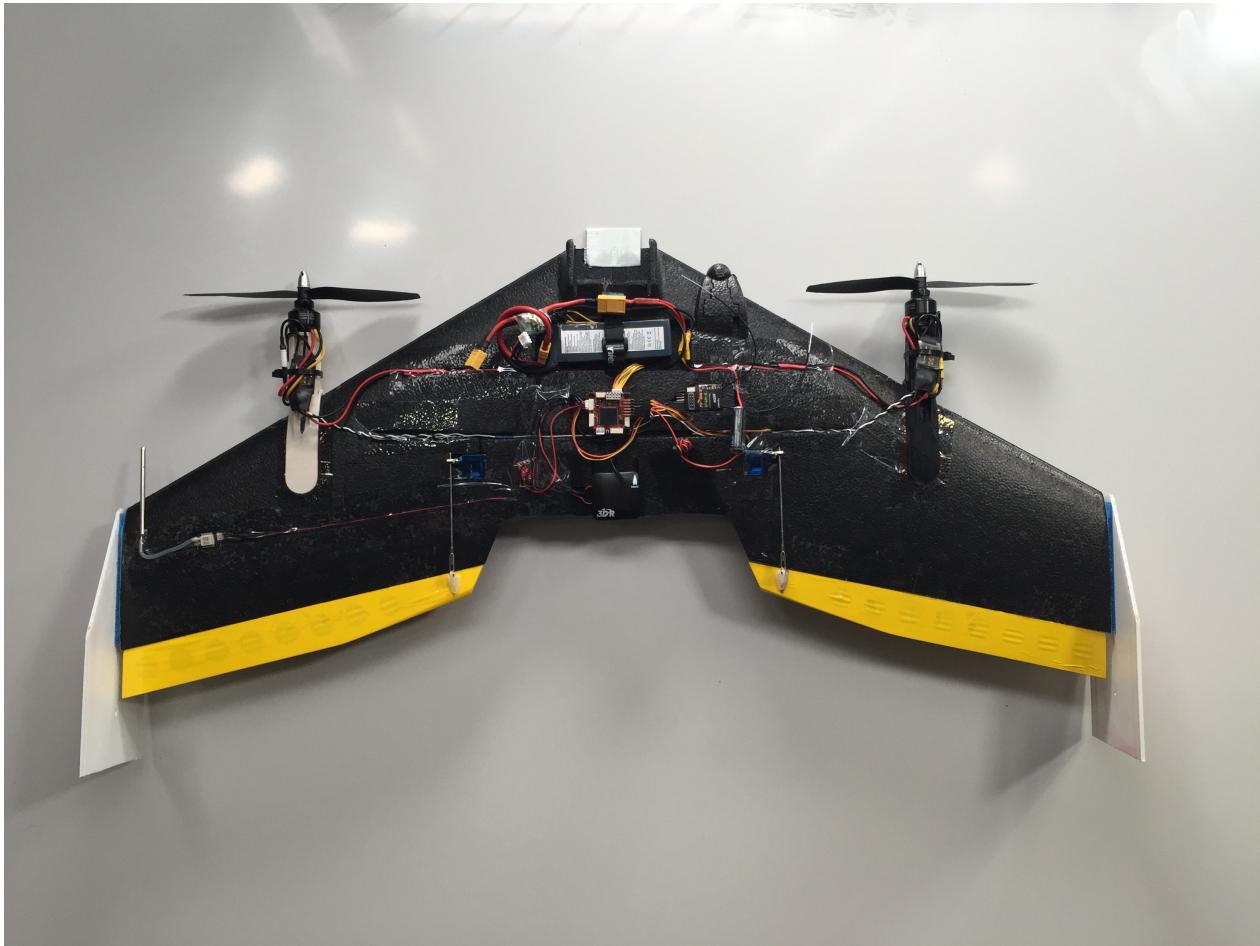
[Video link](#)

Parts List

- TBS Caipirinha Wing ([Eflight store](#))
- Left and right 3D-printed motor mount ([design files](#))
- CW 8045 propeller ([Eflight store](#))
- CCW 8045 propeller ([Eflight store](#))
- 2x 1800 kV 120-180W motors
 - Quanum MT2208 1800 kV
 - ePower 2208
- 2x 20-30S ESC
 - [Eflight store](#)
- BEC (3A, 5-5.3V) (only needed if you are using ESCs which cannot act as a 5V power supply for the output rail)
- 3S 2200 mA LiPo battery
 - Team Orion 3S 11.1V 50 C ([Brack store](#))
- Pixracer autopilot board + power module
- Digital airspeed sensor

Assembly

The picture below shows an example of how a fully assembled Caipiroshka could look like.



In the following some general tips on how to build the vehicle will be given.

Autopilot

Mount the autopilot in the middle close to the CG of the airframe.

Motor mounts

Print the motor mount (2 times) of which the link to the STL file was specified in the part list. Attach one motor mount on each wing side such that the motor axis will be roughly going through the center of the elevons (see picture). In the upper picture the horizontal distance between the two motor mounts is 56cm. Once you have marked the correct position on the wing you can cover the area which will be in contact with the mount with standard transparent tape on both the upper and lower wing side. Then apply a layer of hot glue onto this area and glue the motor mount onto the wing. The reason for having tape in between

the wing surface and the hot glue is that you can very easily remove the motor mount by ripping of the tape from the wing without any damage. This is useful when trying to replace a damaged motor mount.

Motor controllers

The motor controllers can be directly mounted on a flat surface of the motor mounts using glue or a cable binder. You will have to route the power cables to the battery bay. You can use an old soldering iron to melt channels into the foam. Connect the power cables of both motor controllers in the battery bay and solder a plug to the end. This will enable you to connect both the motor controllers to the power module. If you don't have motor controllers which can provide 5V for the output rail of the autopilot then you will have to use an external power supply (BEC).

GPS

The GPS can be mounted in the middle at the very back of the airframe. This helps shifting the weight of the plane to the back since the two motors, a camera and a potentially bigger battery can make it quite nose heavy. Also the large distance to the 12V power cables is beneficial for reducing magnetic interference of the external magnetometer.

Airspeed sensor

Attach the pitot tube close to the outside edge of one of the wing sides. Make sure that the pitot is not effected by the airflow of the propeller. You should be fine if the horizontal distance from the tube to the axis of the motors is larger than the radius of the propeller. Use e.g. an old soldering iron to create a recess for the pitot tube, the tubing and the actual sensor (see picture). Create a channel for routing the cable across the wing to the other components.

Sensor connection to the I2C bus

Both the airspeed sensor and the external magnetometer (located in the gps housing) need to be connected to the I2C bus of the autopilot. Therefore, you will have to use an I2C splitter like the one indicated in the part list. Connect the splitter board with the I2C bus of the autopilot. Then connect both the external magnetometer and the airspeed sensor to the splitter board with a standard I2C cable. In the upper picture the splitter board is located on the left side of the GPS unit.

Elevons

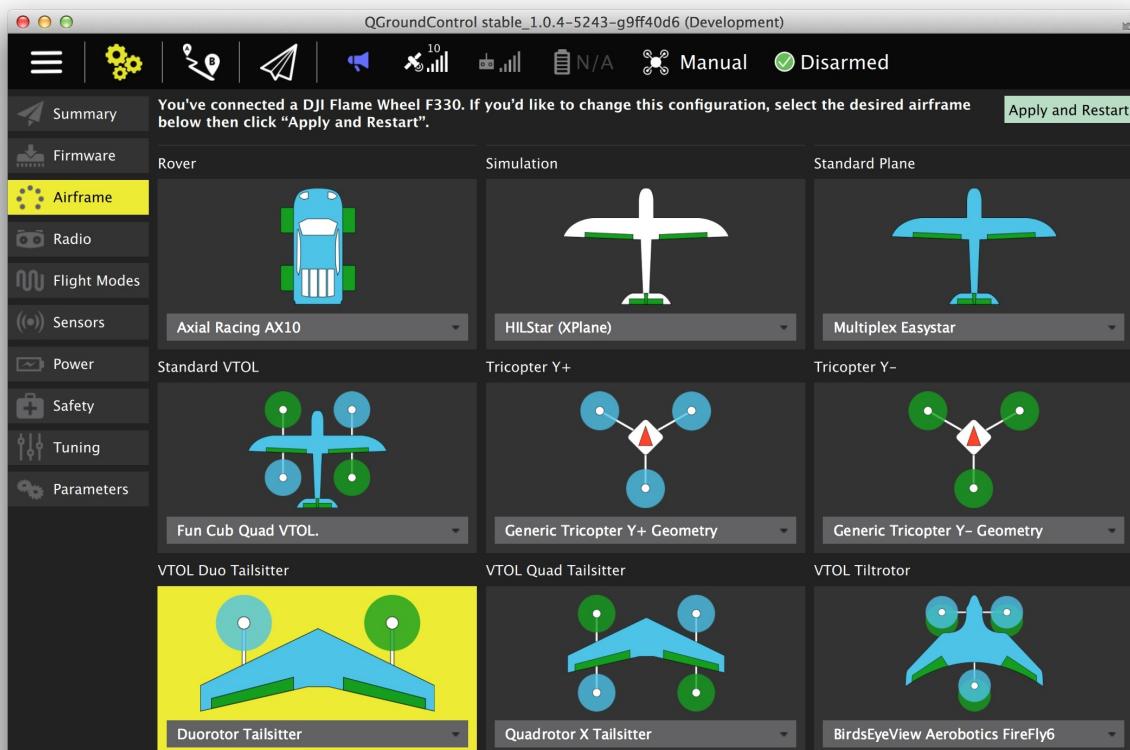
The elevons can be attached to the back side of the wing using transparent tape. You can follow the instructions provided by Team Blacksheep in the build manual for the TBS Caipirinha airframe.

General assembly rules

Before mounting all the components to the wing, use tape to hold them in the approximate position and check if the CG of the wing is in the recommended range specified in the build manual for the TBS Caipirinha. Depending on the additional components you want to have onboard (e.g. GoPro in front or bigger battery) you will need to shift the location of components.

Airframe configuration

Switch to the configuration section in [QGroundControl](#) and select the airframe tab. Scroll down the list to find the VTOL Duorotor Tailsitter icon. Select the `Duorotor Tailsitter` from the drop-down list.



Servo Connections

The descriptions in the table below are referring to the user facing the front of the vehicle when it lies flat on its belly on a table.

Output	Rate	Actuator
MAIN1	400 Hz	Left motor controller
MAIN2	400 Hz	Right motor controller
MAIN3	400 Hz	Empty
MAIN4	400 Hz	Empty
MAIN5	50 Hz	Left aileron servo
MAIN6	50 Hz	Right aileron servo

Experimental Aircraft and General Robots

Work in progress

Companion Computer for Pixhawk class

Interfacing a companion computer (Raspberry Pi, Odroid, Tegra K1) to Pixhawk-family boards always works the same way: They are interfaced using a serial port to `TELEM2`, the port intended for this purpose. The message format on this link is [MAVLink](#).

Pixhawk setup

Set the `SYS_COMPANION` parameter (in the System group) to one of these values.

Info Changing this parameter requires an autopilot reboot to become active.

- `0` to disable MAVLink output on TELEM2 (default)
- `921600` to enable MAVLink output at 921600 baud, 8N1 (recommended)
- `57600` to enable MAVLink output at 57600 baud, 8N1
- `157600` to enable MAVLink in OSD mode at 57600 baud
- `257600` to enable MAVLink in listen-only mode at 57600 baud

Companion computer setup

In order to receive MAVLink, the companion computer needs to run some software talking to the serial port. The most common options are:

- [MAVROS](#) to communicate to ROS nodes
- [C/C++ example code](#) to connect custom code
- [MAVProxy](#) to route MAVLink between serial and UDP

Hardware setup

Wire the serial port according to the instructions below. All Pixhawk serial ports operate at 3.3V and are 5V level compatible.

Warning Many modern companion computers only support 1.8V levels on their hardware UART and can be damaged by 3.3V levels. Use a level shifter. In most cases the accessible hardware serial ports already have some function (modem or console) associated with them and need to be *reconfigured in Linux* before they can be used.

The safe bet is to use an FTDI Chip USB-to-serial adapter board and the wiring below. This always works and is easy to set up.

	TELEM2		FTDI	
1	+5V (red)			DO NOT CONNECT!
2	Tx (out)	5		FTDI RX (yellow) (in)
3	Rx (in)	4		FTDI TX (orange) (out)
4	CTS (in)	6		FTDI RTS (green) (out)
5	RTS (out)	2		FTDI CTS (brown) (in)
6	GND	1		FTDI GND (black)

Software setup on Linux

On Linux the default name of a USB FTDI would be like `\dev\ttyUSB0`. If you have a second FTDI linked on the USB or an Arduino, it will registered as `\dev\ttyUSB1`. To avoid the confusion between the first plugged and the second plugged, we recommend you to create a symlink from `ttyUSBx` to a friendly name, depending on the Vendor and Product ID of the USB device.

Using `lsusb` we can get the vendor and product IDs.

```
$ lsusb

Bus 006 Device 002: ID 0bda:8153 Realtek Semiconductor Corp.
Bus 006 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 005 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 002: ID 05e3:0616 Genesys Logic, Inc.
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 004: ID 2341:0042 Arduino SA Mega 2560 R3 (CDC ACM)
Bus 003 Device 005: ID 26ac:0011
Bus 003 Device 002: ID 05e3:0610 Genesys Logic, Inc. 4-port hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 002: ID 0bda:8176 Realtek Semiconductor Corp. RTL8188CUS 802.11n WLAN Adapter
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

The Arduino is `Bus 003 Device 004: ID 2341:0042 Arduino SA Mega 2560 R3 (CDC ACM)`

The Pixhawk is `Bus 003 Device 005: ID 26ac:0011`

If you do not find your device, unplug it, execute `lsusb`, plug it, execute `lsusb` again and see the added device.

Therefore, we can create a new UDEV rule in a file called `/etc/udev/rules.d/99-pixhawk.rules` with the following content, changing the idVendor and idProduct to yours.

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="2341", ATTRS{idProduct}=="0042", SYMLINK+="ttyArduino"  
SUBSYSTEM=="tty", ATTRS{idVendor}=="26ac", ATTRS{idProduct}=="0011", SYMLINK+="ttyPixhawk"
```

Finally, after a **reboot** you can be sure to know which device is what and put `/dev/ttyPixhawk` instead of `/dev/ttyUSB0` in your scripts.

Be sure to add yourself in the `tty` and `dialout` groups via `usermod` to avoid to have to execute scripts as root.

```
usermod -a -G tty ros-user  
usermod -a -G dialout ros-user
```


Using DroneKit to communicate with PX4

DroneKit helps you create powerful apps for UAVs. These apps run on a UAV's Companion Computer, and augment the autopilot by performing tasks that are both computationally intensive and require a low-latency link (e.g. computer vision).

DroneKit and PX4 are currently working on getting full compatibility. As of DroneKit-python 2.2.0 there is basic support for mission handling and vehicle monitoring.

Setting up DroneKit with PX4

Start by installing DroneKit-python from the current master.

```
git clone https://github.com/dronekit/dronekit-python.git
cd ./dronekit-python
sudo python setup.py build
sudo python setup.py install
```

Create a new python file and import DroneKit, pymavlink and basic modules

```
# Import DroneKit-Python
from dronekit import connect, Command, LocationGlobal
from pymavlink import mavutil
import time, sys, argparse, math
```

Connect to a Mavlink port of your drone or simulation

```
# Connect to the Vehicle
print "Connecting"
connection_string = '127.0.0.1:14540'
vehicle = connect(connection_string, wait_ready=True)
```

Display some basic status information

```
# Display basic vehicle state
print " Type: %s" % vehicle._vehicle_type
print " Armed: %s" % vehicle.armed
print " System status: %s" % vehicle.system_status.state
print " GPS: %s" % vehicle.gps_0
print " Alt: %s" % vehicle.location.global_relative_frame.alt
```

Full mission example

The following python script shows a full mission example using DroneKit and PX4. Mode switching is not yet fully supported from DroneKit, we therefor send our own custom mode switching commands.

```
#####
#####
# @File DroneKitPX4.py
# Example usage of DroneKit with PX4
#
# @author Sander Smeets <sander@droneslab.com>
#
# Code partly based on DroneKit (c) Copyright 2015-2016, 3D Robotics.
#####
#####

# Import DroneKit-Python
from dronekit import connect, Command, LocationGlobal
from pymavlink import mavutil
import time, sys, argparse, math


#####
#####
# Settings
#####
#####

connection_string      = '127.0.0.1:14540'
MAV_MODE_AUTO     = 4
# https://github.com/PX4/Firmware/blob/master/Tools/mavlink_px4.py


# Parse connection argument
parser = argparse.ArgumentParser()
parser.add_argument("-c", "--connect", help="connection string")
args = parser.parse_args()

if args.connect:
    connection_string = args.connect


#####
#####
# Init
#####
#####

# Connect to the Vehicle
print "Connecting"
```

```

vehicle = connect(connection_string, wait_ready=True)

def PX4setMode(mavMode):
    vehicle._master.mav.command_long_send(vehicle._master.target_system, vehicle._master.target_component,
                                           mavutil.mavlink.MAV_CMD_DO_SET_MODE, 0,
                                           mavMode,
                                           0, 0, 0, 0, 0, 0)

def get_location_offset_meters(original_location, dNorth, dEast, alt):
    """
    Returns a LocationGlobal object containing the latitude/longitude `dNorth` and `dE
    ast` metres from the
    specified `original_location`. The returned Location has the same `alt` value
    as `original_location`.

    The function is useful when you want to move the vehicle around specifying locatio
    ns relative to
    the current vehicle position.

    The algorithm is relatively accurate over small distances (10m within 1km) except
    close to the poles.

    For more information see:
    http://gis.stackexchange.com/questions/2951/algorithm-for-offsetting-a-latitude-lo
    ngitude-by-some-amount-of-meters
    """
    earth_radius=6378137.0 #Radius of "spherical" earth
    #Coordinate offsets in radians
    dLat = dNorth/earth_radius
    dLon = dEast/(earth_radius*math.cos(math.pi*original_location.lat/180))

    #New position in decimal degrees
    newlat = original_location.lat + (dLat * 180/math.pi)
    newlon = original_location.lon + (dLon * 180/math.pi)
    return LocationGlobal(newlat, newlon,original_location.alt+alt)

#####
### Listeners
#####

home_position_set = False

#Create a message listener for home position fix
@vehicle.on_message('HOME_POSITION')
def listener(self, name, home_position):
    global home_position_set
    home_position_set = True

```

```
#####
#####
# Start mission example
#####
#####

# wait for a home position lock
while not vehicle.home_onboard:
    print "Waiting for home position..."
    time.sleep(1)

# Display basic vehicle state
print " Type: %s" % vehicle._vehicle_type
print " Armed: %s" % vehicle.armed
print " System status: %s" % vehicle.system_status.state
print " GPS: %s" % vehicle.gps_0
print " Alt: %s" % vehicle.location.global_relative_frame.alt

# Change to AUTO mode
PX4.set_mode(MAV_MODE_AUTO)
time.sleep(1)

# Load commands
cmds = vehicle.commands
cmds.clear()

home = vehicle.location.global_frame

# takeoff to 10 meters
wp = get_location_offset_meters(home, 0, 0, 10);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT, mavutil.mavlink.MA
V_CMD_NAV_TAKEOFF, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

# move 10 meters north
wp = get_location_offset_meters(wp, 10, 0, 0);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT, mavutil.mavlink.MA
V_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

# move 10 meters east
wp = get_location_offset_meters(wp, 0, 10, 0);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT, mavutil.mavlink.MA
V_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

# move 10 meters south
wp = get_location_offset_meters(wp, -10, 0, 0);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT, mavutil.mavlink.MA
V_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
```

```
cmds.add(cmd)

# move 10 meters west
wp = get_location_offset_meters(wp, 0, -10, 0);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT, mavutil.mavlink.MA
V_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

# land
wp = get_location_offset_meters(home, 0, 0, 10);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT, mavutil.mavlink.MA
V_CMD_NAV_LAND, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

# Upload mission
cmds.upload()
time.sleep(2)

# Arm vehicle
vehicle.armed = True

# monitor mission execution
nextwaypoint = vehicle.commands.next
while nextwaypoint < len(vehicle.commands):
    if vehicle.commands.next > nextwaypoint:
        display_seq = vehicle.commands.next+1
        print "Moving to waypoint %s" % display_seq
        nextwaypoint = vehicle.commands.next
    time.sleep(1)

# wait for the vehicle to land
while vehicle.commands.next > 0:
    time.sleep(1)

# Disarm vehicle
vehicle.armed = False
time.sleep(1)

# Close vehicle object before exiting script
vehicle.close()
time.sleep(1)
```


Offboard Control

Warning Offboard control is dangerous. It is the responsibility of the developer to ensure adequate preparation, testing and safety precautions are taken before offboard flights.

The idea behind off-board control is to be able to control the px4 flight stack using software running outside of the autopilot. This is done through the Mavlink protocol, specifically the [SET_POSITION_TARGET_LOCAL_NED](#) and the [SET_ATTITUDE_TARGET](#) messages.

Offboard Control Firmware Setup

There are two things you want to setup on the firmware side before starting offboard development.

1. Map an RC switch to offboard mode activation

To do this, load up the parameters in qGroundcontrol and look for the `RC_MAP_OFFB_SW` parameter to which you can assign the RC channel you want to use to activate offboard mode. It can be useful to map things in such a way that when you fall out of offboard mode you go into position control.

Although this step isn't mandatory since you can activate offboard mode using a MAVLink message. We consider this method much safer.

2. Enable the companion computer interface

Look for the [SYS_COMPANION](#) parameter and set it to either 921600 (Recommended) or 57600. This parameter will activate a MAVLink stream on the Telem2 port with data streams specific to onboard mode with the appropriate baud rate (921600 8N1 or 57600 8N1).

For more information on these data streams, look for "MAVLINK_MODE_ONBOARD" in the [source code](#).

Hardware setup

Usually, there are three ways of setting up offboard communication.

1. Serial radios

1. One connected to a UART port of the autopilot
2. One connected to a ground station computer

Example radios include

- [Lairdtech RM024](#)
- [Digi International XBee Pro](#)



2. On-board processor

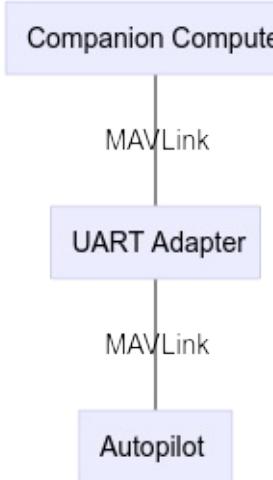
A small computer mounted onto the vehicle connected to the autopilot through a UART to USB adapter. There are many possibilities here and it will depend on what kind of additional on-board processing you want to do in addition to sending commands to the autopilot.

Small low power examples:

- [Odroid C1+ or Odroid XU4](#)
- [Raspberry Pi](#)
- [Intel Edison](#)

Larger high power examples

- [Intel NUC](#)
- [Gigabyte Brix](#)
- [Nvidia Jetson TK1](#)



3. On-board processor and wifi link to ROS *(Recommended)*

A small computer mounted onto the vehicle connected to the autopilot through a UART to USB adapter while also having a WiFi link to a ground station running ROS. This can be any of the computers from the above section coupled with a WiFi adapter. For example, the Intel NUC D34010WYB has a PCI Express Half-Mini connector which can accomodate an [Intel Wifi Link 5000](#) adapter.

```

graph TD
    subgraph Ground Station gnd[ROS Enabled Computer]
        qgc[qGroundControl]
    end
    gnd --- MAVLink1[MAVLink/UDP]
    MAVLink1 --> w[WiFi]
    qgc --- MAVLink2[MAVLink]
    MAVLink2 --> w
    subgraph Vehicle
        comp[Companion Computer]
        uart[UART Adapter]
        autopilot[Autopilot]
    end
    comp --- MAVLink3[MAVLink]
    MAVLink3 --> uart
    uart --- autopilot
    w --- comp
  
```


Raspberry Pi - ROS installation

This is a guide on how to install ROS-indigo on a Raspberry Pi 2 serving as a companion computer for Pixhawk.

Prerequisites

- A working Raspberry Pi with monitor, keyboard, or configured SSH connection
- This guide assumes that you have Raspbian "JESSIE" installed on your RPi. If not: [install it](#) or [upgrade](#) your Raspbian Wheezy to Jessie.

Installation

Follow [this guide](#) for the actual installation of ROS Indigo. Note: Install the "ROS-Comm" variant. The Desktop variant is too heavyweight.

Errors when installing packages

If you want to download packages (e.g. `sudo apt-get install ros-indigo-ros-tutorials`), you might get an error saying: "unable to locate package ros-indigo-ros-tutorials".

If so, proceed as follows: Go to your catkin workspace (e.g. `~/ros_catkin_ws`) and change the name of the packages.

```
$ cd ~/ros_catkin_ws

$ rosinstall_generator ros_tutorials --rosdistro indigo --deps --wet-only --exclude roslisp --tar > indigo-custom_ros.rosinstall
```

Next, update your workspace with wstool.

```
$ wstool merge -t src indigo-custom_ros.rosinstall

$ wstool update -t src
```

Next (still in your workspace folder), source and make your files.

```
$ source /opt/ros/indigo/setup.bash  
$ source devel/setup.bash  
$ catkin_make
```

MAVROS

The [mavros](#) ros package enables MAVLink extendable communication between computers running ROS, MAVLink enabled autopilots, and MAVLink enabled GCS. While MAVRos can be used to communicate with any MAVLink enabled autopilot this documentation will be in the context of enabling communication between the PX4 flight stack and a ROS enabled companion computer.

Installation

MAVROS can be installed either from source or binary. Developers working with ROS are advised to use the source installation.

Binary installation (Debian / Ubuntu)

Since v0.5 that programs available in precompiled debian packages for x86 and amd64 (x86_64). Also v0.9+ exists in ARMv7 repo for Ubuntu armhf. Just use `apt-get` for installation:

```
$ sudo apt-get install ros-indigo-mavros ros-indigo-mavros-extras
```

Source installation

Dependencies

This installation assumes you have a catkin workspace located at `~/catkin_ws`. If you don't create one with:

```
$ mkdir -p ~/catkin_ws/src  
$ cd ~/catkin_ws  
$ catkin init
```

You will be using the ROS python tools `wstool`, `rosinstall`, and `catkin_tools` for this installation. While they may have been installed during your installation of ROS you can also install them with:

```
$ sudo apt-get install python-wstool python-rosinstall-generator python-catkin-tools
```

Note that while the package can be built using `catkin_make` the preferred method is using `catkin_tools` as it is a more versatile and "friendly" build tool.

If this is your first time using `wstool` you will need to initialize your source space with:

```
$ wstool init ~/catkin_ws/src
```

Now you are ready to do the build

```
# 1. get source (upstream - released)
$ rosinstall_generator --upstream mavros | tee /tmp/mavros.rosinstall
    # alternative: latest source
$ rosinstall_generator --upstream-development mavros | tee /tmp/mavros.rosinstall

# 2. get latest released mavlink package
# you may run from this line to update ros-*-*mavlink package
$ rosinstall_generator mavlink | tee -a /tmp/mavros.rosinstall

# 3. Setup workspace & install deps
$ wstool merge -t src /tmp/mavros.rosinstall
$ wstool update -t src
$ rosdep install --from-paths src --ignore-src --rosdistro indigo -y

# finally - build
$ catkin build
```

If you are installing mavros on a raspberry pi, you may get an error related to your os, when running "rosdep install ...". Add "--os=OS_NAME:OS_VERSION" to the rosdep command and replace OS_NAME with your OS name and OS_VERSION with your OS version (e.g. --os=debian:jessie).

MAVROS offboard control example

Offboard control is dangerous. If you are operating on a real vehicle be sure to have a way of gaining back manual control in case something goes wrong.

The following tutorial will run through the basics of offboard control through mavros as applied to an Iris quadcopter simulated in Gazebo. At the end of the tutorial, you should see the same behaviour as in the video below, i.e. a slow takeoff to an altitude of 2 meters.

Code

Create the offb_node.cpp file in your ros package and paste the following inside it:

```
/*
 * @file offb_node.cpp
 * @brief offboard example node, written with mavros version 0.14.2, px4 flight
 * stack and tested in Gazebo SITL
 */

#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <mavros_msgs/CommandBool.h>
#include <mavros_msgs/SetMode.h>
#include <mavros_msgs/State.h>

mavros_msgs::State current_state;
void state_cb(const mavros_msgs::State::ConstPtr& msg){
    current_state = *msg;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "offb_node");
    ros::NodeHandle nh;

    ros::Subscriber state_sub = nh.subscribe<mavros_msgs::State>
        ("mavros/state", 10, state_cb);
    ros::Publisher local_pos_pub = nh.advertise<geometry_msgs::PoseStamped>
        ("mavros/setpoint_position/local", 10);
    ros::ServiceClient arming_client = nh.serviceClient<mavros_msgs::CommandBool>
        ("mavros/cmd/arming");
    ros::ServiceClient set_mode_client = nh.serviceClient<mavros_msgs::SetMode>
        ("mavros/set_mode");

    //the setpoint publishing rate MUST be faster than 2Hz
    ros::Rate rate(20.0);
```

```

// wait for FCU connection
while(ros::ok() && current_state.connected){
    ros::spinOnce();
    rate.sleep();
}

geometry_msgs::PoseStamped pose;
pose.pose.position.x = 0;
pose.pose.position.y = 0;
pose.pose.position.z = 2;

//send a few setpoints before starting
for(int i = 100; ros::ok() && i > 0; --i){
    local_pos_pub.publish(pose);
    ros::spinOnce();
    rate.sleep();
}

mavros_msgs::SetMode offb_set_mode;
offb_set_mode.request.custom_mode = "OFFBOARD";

mavros_msgs::CommandBool arm_cmd;
arm_cmd.request.value = true;

ros::Time last_request = ros::Time::now();

while(ros::ok()){
    if( current_state.mode != "OFFBOARD" &&
        (ros::Time::now() - last_request > ros::Duration(5.0))){
        if( set_mode_client.call(offb_set_mode) &&
            offb_set_mode.response.success){
            ROS_INFO("Offboard enabled");
        }
        last_request = ros::Time::now();
    } else {
        if( !current_state.armed &&
            (ros::Time::now() - last_request > ros::Duration(5.0))){
            if( arming_client.call(arm_cmd) &&
                arm_cmd.response.success){
                ROS_INFO("Vehicle armed");
            }
            last_request = ros::Time::now();
        }
    }
}

local_pos_pub.publish(pose);

ros::spinOnce();
rate.sleep();
}

return 0;

```

```
}
```

Code explanation

```
#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <mavros_msgs/CommandBool.h>
#include <mavros_msgs/SetMode.h>
#include <mavros_msgs/State.h>
```

The `mavros_msgs` package contains all of the custom messages required to operate services and topics provided by the `mavros` package. All services and topics as well as their corresponding message types are documented in the [mavros wiki](#).

```
mavros_msgs::State current_state;
void state_cb(const mavros_msgs::State::ConstPtr& msg){
    current_state = *msg;
}
```

We create a simple callback which will save the current state of the autopilot. This will allow us to check connection, arming and offboard flags.

```
ros::Subscriber state_sub = nh.subscribe<mavros_msgs::State>("mavros/state", 10, state_cb);
ros::Publisher local_pos_pub = nh.advertise<geometry_msgs::PoseStamped>("mavros/setpoint_position/local", 10);
ros::ServiceClient arming_client = nh.serviceClient<mavros_msgs::CommandBool>("mavros/cmd/arming");
ros::ServiceClient set_mode_client = nh.serviceClient<mavros_msgs::SetMode>("mavros/set_mode");
```

We instantiate a publisher to publish the commanded local position and the appropriate clients to request arming and mode change. Note that for your own system, the "mavros" prefix might be different as it will depend on the name given to the node in its launch file.

```
//the setpoint publishing rate MUST be faster than 2Hz
ros::Rate rate(20.0);
```

The px4 flight stack has a timeout of 500ms between two offboard commands. If this timeout is exceeded, the commander will fall back to the last mode the vehicle was in before entering offboard mode. This is why the publishing rate **must** be faster than 2 Hz to also

account for possible latencies. This is also the same reason why it is recommended to enter offboard mode from POSCTL mode, this way if the vehicle drops out of offboard mode it will stop in its tracks and hover.

```
// wait for FCU connection
while(ros::ok() && current_state.connected){
    ros::spinOnce();
    rate.sleep();
}
```

Before publishing anything, we wait for the connection to be established between mavros and the autopilot. This loop should exit as soon as a heartbeat message is received.

```
geometry_msgs::PoseStamped pose;
pose.pose.position.x = 0;
pose.pose.position.y = 0;
pose.pose.position.z = 2;
```

Even though the px4 flight stack operates in the aerospace NED coordinate frame, mavros translates these coordinates to the standard ENU frame and vice-versa. This is why we set z to positive 2.

```
//send a few setpoints before starting
for(int i = 100; ros::ok() && i > 0; --i){
    local_pos_pub.publish(pose);
    ros::spinOnce();
    rate.sleep();
}
```

Before entering offboard mode, you must have already started streaming setpoints otherwise the mode switch will be rejected. Here, 100 was chosen as an arbitrary amount.

```
mavros_msgs::SetMode offb_set_mode;
offb_set_mode.request.custom_mode = "OFFBOARD";
```

We set the custom mode to `OFFBOARD`. A list of [supported modes](#) is available for reference.

```

mavros_msgs::CommandBool arm_cmd;
arm_cmd.request.value = true;

ros::Time last_request = ros::Time::now();

while(ros::ok()){
    if( current_state.mode != "OFFBOARD" &&
        (ros::Time::now() - last_request > ros::Duration(5.0))){
        if( set_mode_client.call(offb_set_mode) &&
            offb_set_mode.response.success){
            ROS_INFO("Offboard enabled");
        }
        last_request = ros::Time::now();
    } else {
        if( !current_state.armed &&
            (ros::Time::now() - last_request > ros::Duration(5.0))){
            if( arming_client.call(arm_cmd) &&
                arm_cmd.response.success){
                ROS_INFO("Vehicle armed");
            }
            last_request = ros::Time::now();
        }
    }

    local_pos_pub.publish(pose);

    ros::spinOnce();
    rate.sleep();
}

```

The rest of the code is pretty self explanatory. We attempt to switch to offboard mode after which we arm the quad to allow it to fly. We space out the service calls by 5 seconds so as to not flood the autopilot with the requests. In the same loop we continue sending the requested pose at the appropriate rate.

This code has been simplified to the bare minimum for illustration purposes. In larger systems, it is often useful to create a new thread which will be in charge of periodically publishing the setpoint.

Using Vision or Motion Capture systems

This page aims at getting a PX4 based system using position data from sources other than GPS (such as motion capture systems like VICON and Optitrack and vision based estimation systems like [ROVIO](#), [SVO](#) or [PTAM](#))

Position estimates can be sent both from an onboard computer as well as from offboard (example : VICON). This data is used to update its local position estimate relative to the local origin. Heading from the vision/motion capture system can also be optionally integrated by the attitude estimator.

The system can then be used for applications such as position hold indoors or waypoint navigation based on vision.

For vision, the mavlink message used to send the pose data is [VISION_POSITION_ESTIMATE](#) and the message for all motion capture systems is [ATT_POS_MOCAP](#) messages.

The mavros ROS-Mavlink interface has default implementations to send these messages. They can also be sent using pure C/C++ code and direct use of the MAVLink() library.

Enabling external pose input

You need to set 2 parameters (from QGroundControl or the NSH shell) to enable or disable vision/mocap usage in the system.

Set the system parameter ``CBRK_NO_VISION`` to 0 to enable vision position integration. Set the system parameter ``ATT_EXT_HDG_M`` to 1 or 2 to enable external heading integration. Setting it to 1 will cause vision to be used, while 2 enables mocap heading use.

OctoMap

The OctoMap library implements a 3D occupancy grid mapping approach. This guide covers the how to use it with [Rotors Simulator](#).

Installation

The installation requires to install ROS, Gazebo and the Rotors Simulator plugin. Follow the [instructions](#) on Rotors Simulator to install.

Next, install The OctoMap library

```
sudo apt-get install ros-indigo-octomap ros-indigo-octomap-mapping  
rosdep install octomap_mapping  
rosmake octomap_mapping
```

Now, open `~/catkin_ws/src/rotors_simulator/rotors_gazebo/CMakeLists.txt` and add the following lines to the bottom of the file

```
find_package(octomap REQUIRED)  
include_directories(${OCTOMAP_INCLUDE_DIRS})  
link_libraries(${OCTOMAP_LIBRARIES})
```

Open `~/catkin_ws/src/rotors_simulator/rotors_gazebo/package.xml` and add the following lines

```
<build_depend>octomap</build_depend>  
<run_depend>octomap</run_depend>
```

Run the following two lines.

The first line changes your default shell editor (which is vim by default) to gedit. This is recommended for users who have little experience with vim, but can otherwise be omitted.

```
export EDITOR='gedit'  
rosed octomap_server octomap_tracking_server.launch
```

and change the two following lines

```
<param name="frame_id" type="string" value="map" />
...
<!--remap from="cloud_in" to="/rgbdslam/batch_clouds" /-->
```

to

```
<param name="frame_id" type="string" value="world" />
...
<remap from="cloud_in" to="/firefly/vi_sensor/camera_depth/depth/points" />
```

Running the Simulation

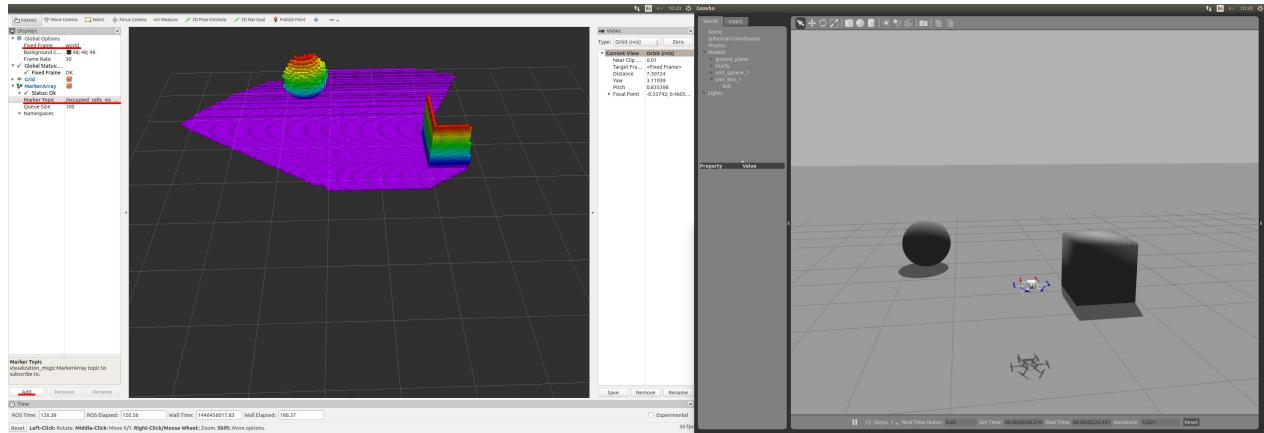
Now run the three following lines, in three separate terminal windows. This opens up Gazebo, Rviz and an octomap server.

```
roslaunch rotors_gazebo mav_hovering_example_with_vi_sensor.launch mav_name:=firefly
rviz
roslaunch octomap_server octomap_tracking_server.launch
```

In Rviz, change the field 'Fixed Frame' from 'map' to 'world' in the top left of the window. Now click the add button in the bottom left and select MarkerArray. Then double click the MarkerArray and change 'Marker Topic' from '/free_cells_vis_array' to '/occupied_cells_vis_array'

Now you should see a part of the floor.

In the Gazebo window, insert a cube in front of the red rotors and you should see it in Rviz.

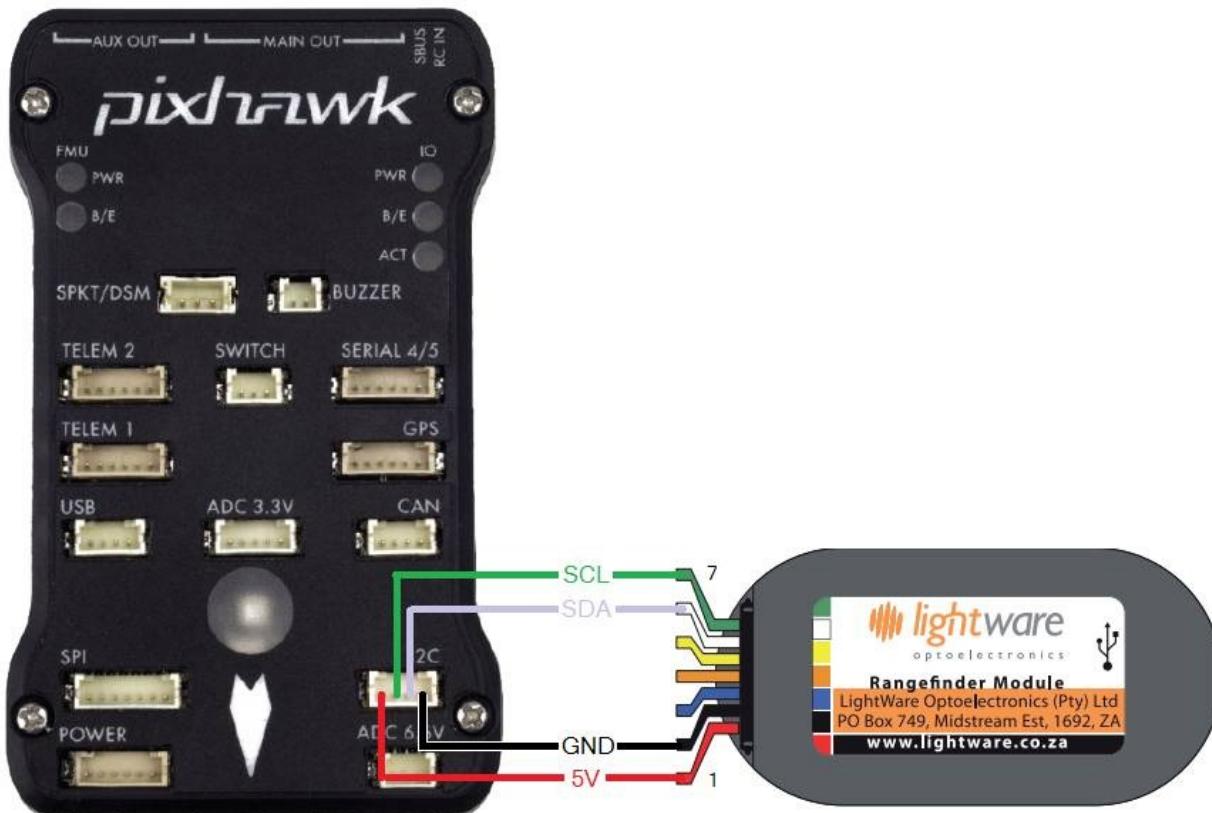


Lightware SF1XX lidar setup

This page shows you how to set up one of following lidars:

1. SF10/a
2. SF10/b
3. SF10/c
4. SF11/c

Driver supports only i2c connection.



Configuring lidar

You should connect to sensor via usb (it has internal usb to serial converter), run terminal, press `space` and check that i2c address equal to `0x66`. Newer sensor versions already have `0x66` preconfigured. Older have `0x55` which conflicts with `rgbled` module.

Configuring PX4

Use the `SENS_EN_SF1XX` parameter to select the lidar model and then reboot.

- 0 lidar disabled
- 1 SF10/a
- 2 SF10/b
- 3 SF10/c
- 4 SF11/c

UAVCAN Introduction



UAVCAN is an onboard network which allows the autopilot to connect to avionics. It supports hardware like:

- Motor controllers
 - [Pixhawk ESC](#)
 - [SV2740 ESC](#)
- Airspeed sensors
 - [Thiemar airspeed sensor](#)
- GNSS receivers for GPS and GLONASS
 - [Zubax GNSS](#)

In contrast to hobby-grade devices it uses rugged, differential signalling and supports firmware upgrades over the bus. All motor controllers provide status feedback and implement field-oriented-control (FOC).

Upgrading Node Firmware

The PX4 middleware will automatically upgrade firmware on UAVCAN nodes if the matching firmware is supplied. The process and requirements are described on the [UAVCAN Firmware](#) page.

Enumerating and Configuring Motor Controllers

The ID and rotational direction of each motor controller can be assigned after installation in a simple setup routine: [UAVCAN Node Enumeration](#). The routine can be started by the user through QGroundControl.

Useful links

- [Homepage](#)

- Specification
- Implementations and tutorials

UAVCAN Bootloader Installation

UAVCAN devices typically ship with a bootloader pre-installed. Do not follow the instructions in this section unless you are developing UAVCAN devices.

Overview

The PX4 project includes a standard UAVCAN bootloader for STM32 devices.

The bootloader occupies the first 8–16 KB of flash, and is the first code executed on power-up. Typically, the bootloader performs low-level device initialization, automatically determines the CAN bus baud rate, acts as a UAVCAN dynamic node ID client to obtain a unique node ID, and waits for confirmation from the flight controller before proceeding with application boot.

This process ensures that a UAVCAN device can recover from invalid or corrupted application firmware without user intervention, and also permits automatic firmware updates.

Prerequisites

Installing or updating the UAVCAN bootloader requires:

- An SWD or JTAG interface (depending on device), for example the [BlackMagic Probe](#) or the [ST-Link v2](#);
- An adapter cable to connect your SWD or JTAG interface to the UAVCAN device's debugging port;
- A [supported ARM toolchain](#).

Device Preparation

If you are unable to connect to your device using the instructions below, it's possible that firmware already on the device has disabled the MCU's debug pins. To recover from this, you will need to connect your interface's NRST or nSRST pin (pin 15 on the standard ARM 20-pin connector) to your MCU's NRST pin. Obtain your device schematics and PCB layout or contact the manufacturer for details.

Installation

After compiling or obtaining a bootloader image for your device (refer to device documentation for details), the bootloader must be copied to the beginning of the device's flash memory.

The process for doing this depends on the SWD or JTAG interface used.

BlackMagic Probe

Ensure your BlackMagic Probe [firmware is up to date](#).

Connect the probe to your UAVCAN device, and connect the probe to your computer.

Identify the probe's device name. This will typically be `/dev/ttyACM<x>` or `/dev/ttyUSB<x>`.

Power up your UAVCAN device, and run:

```
arm-none-eabi-gdb /path/to/your/bootloader/image.elf
```

At the `gdb` prompt, run:

```
target extended /dev/ttyACM0
monitor connect_srst enable
monitor swdp_scan
attach 1
set mem inaccessible-by-default off
load
run
```

If `monitor swdp_scan` returns an error, ensure your wiring is correct, and that you have an up-to-date version of the BlackMagic firmware.

ST-Link v2

Ensure you have a recent version—at least 0.9.0—of [OpenOCD](#).

Connect the ST-Link to your UAVCAN device, and connect the ST-Link to your computer.

Power up your UAVCAN device, and run:

```
openocd -f /path/to/your/openocd.cfg &
arm-none-eabi-gdb /path/to/your/bootloader/image.elf
```

At the `gdb` prompt, run:

```
target extended-remote localhost:3333
monitor reset halt
set mem inaccessible-by-default off
load
run
```

Segger J-Link Debugger

Connect the JLink Debugger to your UAVCAN device, and connect the JLink Debugger to your computer.

Power up your UAVCAN device, and run:

```
JLinkGDBServer -select USB=0 -device STM32F446RE -if SWD-DP -speed 20000 -vd
```

Open a second terminal, navigate to the directory that includes the px4esc_1_6-bootloader.elf for the esc and run:

```
arm-none-eabi-gdb px4esc_1_6-bootloader.elf
```

At the `gdb` prompt, run:

```
tar ext :2331 load
```

Erasing Flash with SEGGER JLink Debugger

As a recovery method it may be useful to erase flash to factory defaults such that the firmware is using the default parameters. Go to the directory of your SEGGER installation and launch JLinkExe, then run:

```
device <name-of-device>
erase
```

Replace `<name-of-device>` with the name of the microcontroller, e.g. STM32F446RE for the Pixhawk ESC 1.6 or STM32F302K8 for the SV2470VC ESC.

UAVCAN Firmware Upgrading

Vectorcontrol ESC Codebase (Pixhawk ESC 1.6 and S2740VC)

Download the ESC code:

```
git clone https://github.com/thiemar/vectorcontrol  
cd vectorcontrol
```

Flashing the UAVCAN Bootloader

Before updating firmware via UAVCAN, the Pixhawk ESC 1.6 requires the UAVCAN bootloader be flashed. To build the bootloader, run:

```
make clean && BOARD=px4esc_1_6 make -j8
```

After building, the bootloader image is located at `firmware/px4esc_1_6-bootloader.bin`, and the OpenOCD configuration is located at `openocd_px4esc_1_6.cfg`. Follow [these instructions](#) to install the bootloader on the ESC.

Compiling the Main Binary

```
BOARD=s2740vc_1_0 make && BOARD=px4esc_1_6 make
```

This will build the UAVCAN node firmware for both supported ESCs. The firmware images will be located at `com.thiemar.s2740vc-v1-1.0-1.0.<git hash>.bin` and `org.pixhawk.px4esc-v1-1.6-1.0.<git hash>.binn`.

Sapog Codebase (Pixhawk ESC 1.4)

Download the Sapog codebase:

```
git clone https://github.com/PX4/sapog  
cd sapog  
git submodule update --init --recursive
```

Flashing the UAVCAN Bootloader

Before updating firmware via UAVCAN, the Pixhawk ESC 1.4 requires the UAVCAN bootloader be flashed. The bootloader can be built as follows:

```
cd bootloader  
make clean && make -j8  
cd ..
```

The bootloader image is located at `bootloader/firmware/bootloader.bin`, and the OpenOCD configuration is located at `openocd.cfg`. Follow [these instructions](#) to install the bootloader on the ESC.

Compiling the Main Binary

```
cd firmware  
make sapog.image
```

The firmware image will be located at `firmware/build/org.pixhawk.sapog-v1-1.0.<xxxxxxxx>.bin`, where `<xxxxxxxx>` is an arbitrary sequence of numbers and letters.

Zubax GNSS

Please refer to the [project page](#) to learn how to build and flash the firmware. Zubax GNSS comes with a UAVCAN-capable bootloader, so its firmware can be updated in a uniform fashion via UAVCAN as described below.

Firmware Installation on the Autopilot

The UAVCAN node file names follow a naming convention which allows the Pixhawk to update all UAVCAN devices on the network, regardless of manufacturer. The firmware files generated in the steps above must therefore be copied to the correct locations on an SD card or the PX4 ROMFS in order for the devices to be updated.

The convention for firmware image names is:

```
<uavcan name>-<hw version major>.<hw version minor>-<sw version major>.<sw version minor>.<version hash>.bin
```

e.g. `com.thiemar.s2740vc-v1-1.0-1.0.68e34de6.bin`

However, due to space/performance constraints (names may not exceed 28 characters), the UAVCAN firmware updater requires those filenames to be split and stored in a directory structure like the following:

```
/fs/microsd/fw/<node name>/<hw version major>.<hw version minor>/<hw name>-<sw version major>.<sw version minor>.<git hash>.bin
```

e.g. s2740vc-v1-1.0.68e34de6.bin

The ROMFS-based updater follows that pattern, but prepends the file name with _ so you add the firmware in:

```
/etc/uavcan/fw/<device name>/<hw version major>.<hw version minor>/_<hw name>-<sw version major>.<sw version minor>.<git hash>.bin
```

Placing the binaries in the PX4 ROMFS

The resulting final file locations are:

- S2740VC ESC: ROMFS/px4fmu_common/uavcan/fw/com.thiemar.s2740vc-v1/1.0/_s2740vc-v1-1.0.<git hash>.bin
- Pixhawk ESC 1.6: ROMFS/px4fmu_common/uavcan/fw/org.pixhawk.px4esc-v1/1.6/_px4esc-v1-1.6.<git hash>.bin
- Pixhawk ESC 1.4: `ROMFS/px4fmu_common/uavcan/fw/org.pixhawk.sapog-v1/1.4/_sapog-v1-1.4..bin`
- Zubax GNSS v1: ROMFS/px4fmu_common/uavcan/fw/com.zubax.gnss/1.0/gnss-1.0.<git has>.bin
- Zubax GNSS v2: ROMFS/px4fmu_common/uavcan/fw/com.zubax.gnss/2.0/gnss-2.0.<git has>.bin

Note that the ROMFS/px4fmu_common directory will be mounted to /etc on Pixhawk.

Starting the Firmware Upgrade process

When using the [PX4 Flight Stack](concept-flight-stack.md), enable UAVCAN in the 'Power Config' section and reboot the system before attempting an UAVCAN firmware upgrade.

Alternatively UAVCAN firmware upgrading can be started manually on NSH via:

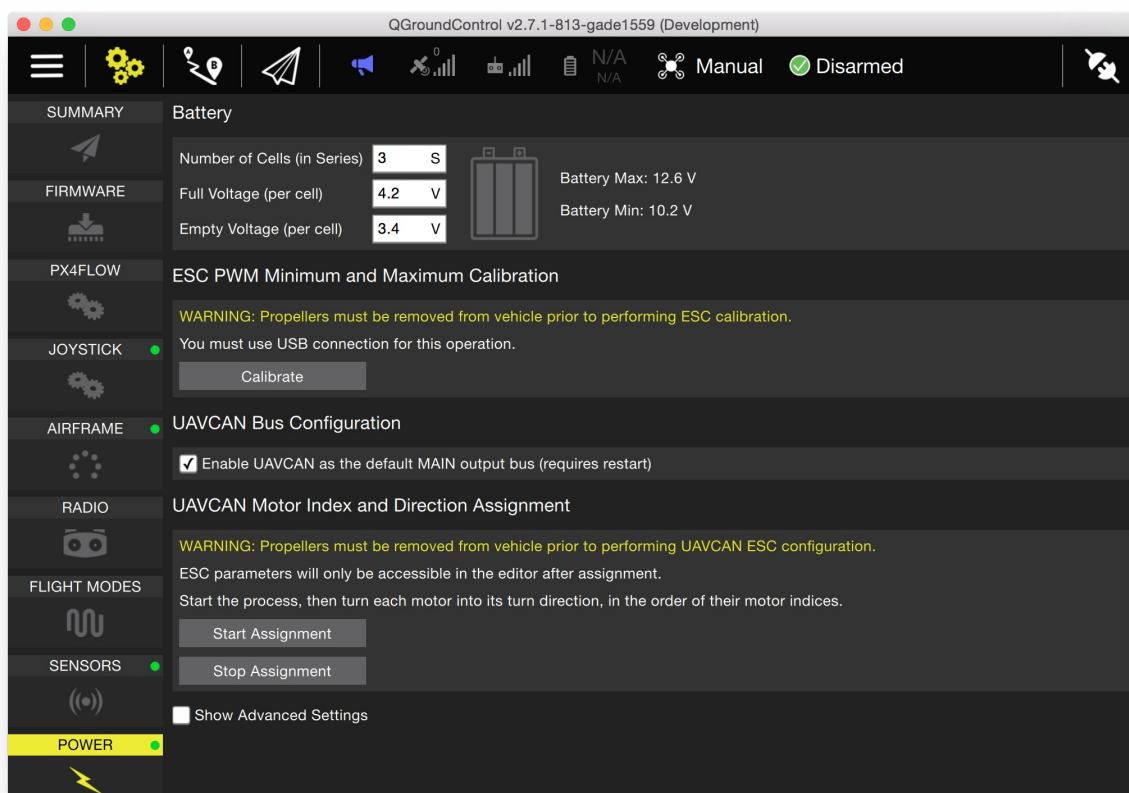
```
uavcan start  
uavcan start fw
```


UAVCAN Enumeration and Configuration

Enable UAVCAN as the default motor output bus by ticking the 'Enable UAVCAN' checkbox as shown below. Alternatively the UAVCAN_ENABLE parameter can be set to '3' in the QGroundControl parameter editor. Set it to '2' to enable CAN, but leave motor outputs on PWM.

Use [QGroundControl](#) and switch to the Setup view. Select the Power Configuration on the left. Click on the 'start assignment' button.

After the first beep, turn the propeller on the first ESC swiftly into the correct turn direction. The ESCs will all beep each time one is enumerated. Repeat this step for all motor controllers in the order as shown on the [motor map](#). This step has to be performed only once and does not need to be repeated after firmware upgrades.



uLanding Radar

The uLanding radar is a product from [Aerotenna](#) and can be used to measure distance to an object.

Enable the driver for your hardware

Currently, this radar device is supported by any hardware which runs the OS NuttX and which can offer a serial port for the interface. Since flash space is small on some hardware you may have to enable building the driver for your target yourself. To do so add the following line to the cmake config file which corresponds to the target you want to build for:

```
drivers/ulanding
```

All config files are located [here](#).

Start the driver

You will have to tell the system to start the driver for the radar during system startup. You can simply add the following line to an [extras.txt](#) file located on your SD card.

```
ulanding_radar start /dev/serial_port
```

In the upper command you will have to replace the last argument with the serial port you have connected the hardware to. If you don't specify any port the driver will use /dev/ttyS2 which is the TELEM2 port on Pixhawk.

Warning

If you are connecting the radar device to TELEM2 then make sure to set the parameter SYS_COMPANION to 0. Otherwise the serial port will be used by another application and you will get unexpected behaviour.

Frequently Asked Questions

Build Errors

Flash Overflow

Use the FMUv4 architecture to obtain double the flash. The first available board from this generation is the [Pixracer](<http://dev.px4.io/hardware-pixracer.html>).

The amount of code that can be loaded onto a board is limited by the amount of flash memory it has. When adding additional modules or code its possible that the addition exceeds the flash memory. This will result in a "flash overflow". The upstream version will always build, but depending on what a developer adds it might overflow locally.

```
region `flash' overflowed by 12456 bytes
```

To remedy it, either use more recent hardware or remove modules from the build which are not essential to your use case. The configurations are stored [here](#). To remove a module, just comment it out:

```
#drivers/trone
```

USB Errors

The upload never succeeds

On Ubuntu, deinstall the modem manager:

```
sudo apt-get remove modemmanager
```

PX4 System Console

The system console allows low-level access to the system, debug output and analysis of the system boot process. The most convenient way to connect it is by using a [Dronecode probe](#), but a plain FTDI cable can be used as well.

System Console vs. Shell

There are multiple shells, but only one console: The system console is the location where all boot output (and applications auto-started on boot) is printed.

- System console (first shell): Hardware serial port
- Additional shells: Pixhawk on USB (e.g. lists as /dev/tty.usbmodem1 on Mac OS)

Info USB shell: To just run a few quick commands or test an application connecting to the USB shell is sufficient. The Mavlink shell can be used for this, see below. The hardware serial console is only needed for boot debugging or when USB should be used for MAVLink to connect a [GCS](#).

Snapdragon Flight: Wiring the Console

The developer kit comes with a breakout board with three pins to access the console. Connect the bundled FTDI cable to the header and the breakout board to the expansion connector.

Pixracer / Pixhawk v3: Wiring the Console

Connect the 6-pos JST SH 1:1 cable to the Dronecode probe or connect the individual pins of the cable to a FTDI cable like this:

Pixracer / Pixhawk v3		FTDI	
1	+5V (red)		N/C
2	UART7 Tx	5	FTDI RX (yellow)
3	UART7 Rx	4	FTDI TX (orange)
4	SWDIO		N/C
5	SWCLK		N/C
6	GND	1	FTDI GND (black)

Pixhawk v1: Wiring the Console

The system console can be accessed through the Dronecode probe or an FTDI cable. Both options are explained in the section below.

Connecting via Dronecode Probe

Connect the 6-pos DF13 1:1 cable on the [Dronecode probe](#) to the SERIAL4/5 port of Pixhawk.

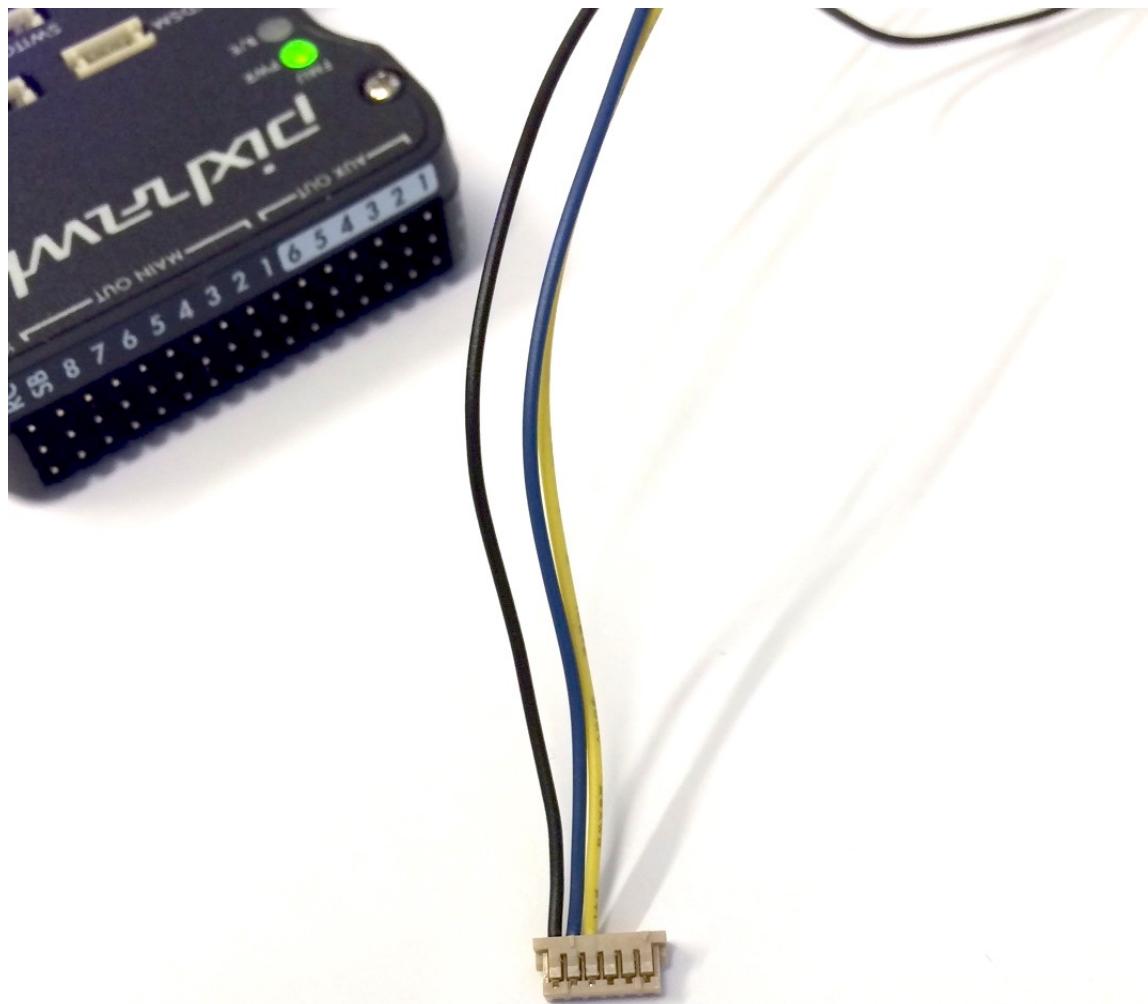


Connecting via FTDI 3.3V Cable

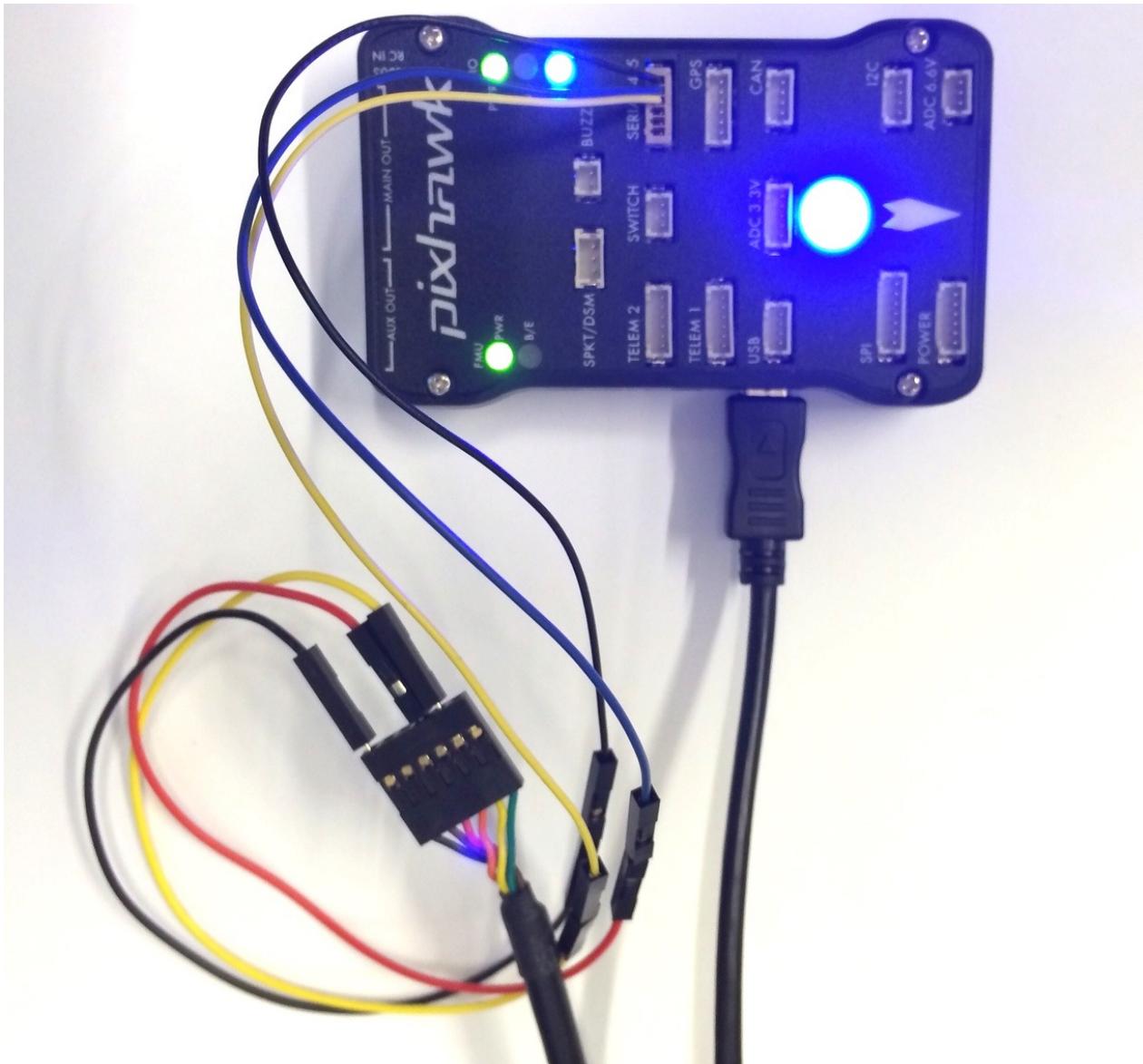
If no Dronecode probe is at hand an FTDI 3.3V (Digi-Key: [768-1015-ND](#)) will do as well.

Pixhawk 1/2		FTDI	
1	+5V (red)		N/C
2	S4 Tx		N/C
3	S4 Rx		N/C
4	S5 Tx	5	FTDI RX (yellow)
5	S5 Rx	4	FTDI TX (orange)
6	GND	1	FTDI GND (black)

The connector pinout is shown in the figure below.



The complete wiring is shown below.



Opening the Console

After the console connection is wired up, use the default serial port tool of your choice or the defaults described below:

Linux / Mac OS: Screen

Install screen on Ubuntu (Mac OS already has it installed):

```
sudo apt-get install screen
```

- Serial: Pixhawk v1 / Pixracer use 57600 baud
- Serial: Snapdragon Flight uses 115200 baud

Connect screen at BAUDRATE baud, 8 data bits, 1 stop bit to the right serial port (use `ls /dev/tty*` and watch what changes when unplugging / replugging the USB device). Common names are `/dev/ttyUSB0` and `/dev/ttyACM0` for Linux and `/dev/tty.usbserial-ABCBD` for Mac OS.

```
screen /dev/ttyXXX BAUDRATE 8N1
```

Windows: PuTTY

Download [PuTTY](#) and start it.

Then select 'serial connection' and set the port parameters to:

- 57600 baud
- 8 data bits
- 1 stop bit

Getting Started on the Console

Type `ls` to view the local file system, type `free` to see the remaining free RAM. The console will also display the system boot log when power-cycling the board.

```
nsh> ls  
nsh> free
```

MAVLink Shell

For NuttX-based systems (Pixhawk, Pixracer, ...), the nsh console can also be accessed via mavlink. This works via serial link or WiFi (UDP/TCP). Make sure that QGC is not running, then start the shell with e.g. `./Tools/mavlink_shell.py /dev/ttyACM0` (in the Firmware source). Use `-h` to get a description of all available arguments. You may first have to install the dependencies with `sudo pip install pymavlink pyserial`.

Snapdragon DSP Console

When you are connected to your Snapdragon board via usb you have access to the px4 shell on the posix side of things. The interaction with the DSP side (QuRT) is enabled with the `qsHELL` posix app and its QuRT companion.

With the Snapdragon connected via USB, open the mini-dm to see the output of the DSP:

```
 ${HEXAGON_SDK_ROOT}/tools/debug/mini-dm/Linux_Debug/mini-dm
```

Note: Alternatively, especially on Mac, you can also use [nano-dm](#).

Run the main app on the linaro side:

```
cd /home/linaro  
./px4 px4.config
```

You can now use all apps loaded on the DSP from the linaro shell with the following syntax:

```
pxh> qshell command [args ...]
```

For example, to see the available QuRT apps:

```
pxh> qshell list_tasks
```

The output of the executed command is displayed on the minidm.

System Startup

The PX4 boot is controlled by shell scripts in the [ROMFS/px4fmu_common/init.d](#) folder.

All files starting with a number and underscore (e.g. `10000_airplane`) are canned airframe configurations. They are exported at build-time into an `airframes.xml` file which is parsed by [QGroundControl](#) for the airframe selection UI. Adding a new configuration is covered [here](#).

The remaining files are part of the general startup logic, and the first executed file is the `rcS` script, which calls all other scripts.

Debugging the System Boot

A failure of a driver or software component can lead to an aborted boot.

An incomplete boot often materializes as missing parameters in the ground control stations, because the non-started applications did not initialize their parameters.

The right approach to debug the boot sequence is to connect the [system console](#) and power-cycle the board. The resulting boot log has detailed information about the boot sequence and should contain hints why the boot aborted.

Common boot failure causes

- A required sensor failed to start
- For custom applications: The system was out of RAM. Run the `free` command to see the amount of free RAM.
- A software fault or assertion resulting in a stack trace

Replacing the System Startup

In most cases customizing the default boot is the better approach, which is documented below. If the complete boot should be replaced, create a file `/fs/microsd/etc/rc.txt`, which is located in the `etc` folder on the microSD card. If this file is present nothing in the system will be auto-started.

Customizing the System Startup

The best way to customize the system startup is to introduce a [new airframe configuration](#). If only tweaks are wanted (like starting one more application or just using a different mixer) special hooks in the startup can be used.

The system boot files are UNIX FILES which require UNIX LINE ENDINGS. If editing on Windows use a suitable editor.

There are three main hooks. Note that the root folder of the microsd card is identified by the path `/fs/microsd`.

- `/fs/microsd/etc/config.txt`
- `/fs/microsd/etc/extras.txt`
- `/fs/microsd/mixers/NAME_OF_MIXER`

Customizing the Configuration (`config.txt`)

The `config.txt` file is loaded after the main system has been configured and *before* it is booted and allows to modify shell variables.

Starting additional applications

The `extras.txt` can be used to start additional applications after the main system boot. Typically these would be payload controllers or similar optional custom components.

Starting a custom mixer

By default the system loads the mixer from `/etc/mixers`. If a file with the same name exists in `/fs/microsd/etc/mixers` this file will be loaded instead. This allows to customize the mixer file without the need to recompile the Firmware.

Parameters & Configurations

The PX4 platform uses the param subsystem (a flat table of float and int32_t values) and text files (for mixers and startup scripts) to store its configuration.

The [system startup](#) and how [airframe configurations](#) work are detailed on other pages. This section discusses the param subsystem in detail

Commandline usage

The PX4 [system console](#) offers the `param` tool, which allows to set parameters, read their value, save them and export and restore to and from files.

Getting and Setting Parameters

The `param show` command lists all system parameters:

```
param show
```

To be more selective a partial parameter name with wildcard can be used:

```
nsh> param show RC_MAP_A*
Symbols: x = used, + = saved, * = unsaved
x   RC_MAP_AUX1 [359,498] : 0
x   RC_MAP_AUX2 [360,499] : 0
x   RC_MAP_AUX3 [361,500] : 0
x   RC_MAP_ACRO_SW [375,514] : 0

723 parameters total, 532 used.
```

Exporting and Loading Parameters

The standard `save` command will store the parameters in the current default file:

```
param save
```

If provided with an argument, it will store the parameters instead to this new location:

```
param save /fs/microsd/vtol_param_backup
```

There are two different commands to load parameters: `param load` will load a file and replace the current parameters with what is in this file, resulting in a 1:1 copy of the state that was present when these parameters were stored. `param import` is more subtle: It will only change parameter values which have been changed from the default. This is great to e.g. initially calibrate a board (without configuring it further), then importing the calibration data without overwriting the rest of the system configuration.

Overwrite the current parameters:

```
param load /fs/microsd/vtol_param_backup
```

Merge the current parameters with stored parameters (stored values which are non-default take precedence):

```
param import /fs/microsd/vtol_param_backup
```

C / C++ API

There is also a C and a separate C++ API which can be used to access parameter values.

Discuss param C / C++ API.

```
int32_t param = 0;  
param_get(param_find("PARAM_NAME"), &param);
```

Parameter Meta Data

PX4 uses an extensive parameter meta data system to drive the user-facing presentation of parameters. Correct meta data is critical for good user experience in the ground station.

A typical parameter meta data section will look like this:

```
/**  
 * Pitch P gain  
 *  
 * Pitch proportional gain, i.e. desired angular speed in rad/s for error 1 rad.  
 *  
 * @unit 1/s  
 * @min 0.0  
 * @max 10  
 * @decimal 2  
 * @increment 0.0005  
 * @reboot_required true  
 * @group Multicopter Attitude Control  
 */  
PARAM_DEFINE_FLOAT(MC_PITCH_P, 6.5f);
```

Where each line has this use:

```
/**  
 * <title>  
 *  
 * <longer description, can be multi-line>  
 *  
 * @unit <the unit, e.g. m for meters>  
 * @min <the minimum sane value. Can be overridden by the user>  
 * @max <the maximum sane value. Can be overridden by the user>  
 * @decimal <the minimum sane value. Can be overridden by the user>  
 * @increment <the "ticks" in which this value will increment in the UI>  
 * @reboot_required true <add this if changing the param requires a system restart>  
 * @group <a title for parameters which form a group>  
 */  
PARAM_DEFINE_FLOAT(MC_PITCH_P, 6.5f);
```

Embedded Debugging

The autopilots running PX4 support debugging via GDB or LLDB.

Identifying large memory consumers

The command below will list the largest static allocations:

```
arm-none-eabi-nm --size-sort --print-size --radix=dec build_px4fmu-v2_default/src/firmware/nuttx/firmware_nuttx | grep "[bBdD]"
```

This NSH command provides the remaining free memory:

```
free
```

And the top command shows the stack usage per application:

```
top
```

Stack usage is calculated with stack coloring and thus is not the current usage, but the maximum since the start of the task.

Heap allocations

Dynamic heap allocations can be traced on POSIX in SITL with [gperftools](#). Once installed, it can be used with:

- Run jmavsim: `./Tools/jmavsim_run.sh`
- Then:

```
cd build_posix_sitl_default/tmp
export HEAPPROFILE=/tmp/heaprofile.hprof
env LD_PRELOAD=/lib64/libtcmalloc.so ./src/firmware/posix/px4 posix-configs/SITL/init/lpe/iris
pprof --pdf ./src/firmware/posix/px4 /tmp/heaprofile.hprof.0001.heap > heap.pdf
```

It will generate a pdf with a graph of the heap allocations. The numbers in the graph will all be zero, because they are in MB. Just look at the percentages instead. They show the live memory (of the node and the subtree), meaning the memory that was still in use at the end.

If it does not generate heap dumps while running the `px4` app you might need to change the settings of the profiler. On some systems it is necessary to set an interval time when to write the dumps:

```
# Specify interval in seconds
export HEAP_PROFILE_TIME_INTERVAL=10
```

See the [gperftools docs](#) for more information.

Sending MAVLink debug key / value pairs

The code for this tutorial is available here:

- [Debug Tutorial Code](#)
- [Enable the tutorial app](#) by uncommenting / enabling the mavlink debug app in the config of your board

All required to set up a debug publication is this code snippet. First add the header file:

```
#include <uORB/uORB.h>
#include <uORB/topics/debug_key_value.h>
```

Then advertise the debug value topic (one advertisement for different published names is sufficient). Put this in front of your main loop:

Debugging Hard Faults in NuttX

A hard fault is a state when the operating system detects that it has no valid instructions to execute. This is typically the case when key areas in RAM have been corrupted. A typical scenario is when incorrect memory access smashed the stack and the processor sees that the address in memory is not a valid address for the microprocessors's RAM.

- NuttX maintains two stacks: The IRQ stack for interrupt processing and the user stack
- The stack grows downward. So the highest address in the example below is 0x20021060, the size is 0x11f4 (4596 bytes) and consequently the lowest address is 0x2001fe6c.

```

Assertion failed at file:armv7-m/up_hardfault.c line: 184 task: ekf_att_pos_estimator
sp:      20003f90
IRQ stack:
  base: 20003fdc
  size: 000002e8
20003f80: 080d27c6 20003f90 20021060 0809b8d5 080d288c 000000b8 08097155 00000010
20003fa0: 20003ce0 00000003 00000000 0809bb61 0809bb4d 080a6857 e000ed24 080a3879
20003fc0: 00000000 2001f578 080ca038 000182b8 20017cc0 0809bad1 20020c14 00000000
sp:      20020ce8
User stack:
  base: 20021060
  size: 000011f4
20020ce0: 60000010 2001f578 2001f578 080ca038 000182b8 0808439f 2001fb88 20020d4c
20020d00: 20020d44 080a1073 666b655b 65686320 205d6b63 6f6c6576 79746963 76696420
20020d20: 65747265 63202c64 6b636568 63636120 63206c65 69666e6f 08020067 0805c4eb
20020d40: 080ca9d4 0805c21b 080ca1cc 080ca9d4 385833fb 38217db9 00000000 080ca964
20020d60: 080ca980 080ca9a0 080ca9bc 080ca9d4 080ca9fc 080caa14 20022824 00000002
20020d80: 2002218c 0806a30f 08069ab2 81000000 3f7ffffec 00000000 3b4ae00c 3b12eaa6
20020da0: 00000000 00000000 080ca010 4281fb70 20020f78 20017cc0 20020f98 20017cdc
20020dc0: 2001ee0c 0808d7ff 080ca010 00000000 3f800000 00000000 080ca020 3aa35c4e
20020de0: 3834d331 00000000 01010101 00000000 01010001 000d4f89 000d4f89 000f9fda
20020e00: 3f7d8df4 3bac67ea 3ca594e6 be0b9299 40b643aa 41ebe4ed bcc04e1b 43e89c96
20020e20: 448f3bc9 c3c50317 b4c8d827 362d3366 b49d74cf ba966159 00000000 00000000
20020e40: 3eb4da7b 3b96b9b7 3eedad6a 00000000 00000000 00000000 00000000 00000000
20020e60: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
20020e80: 00000016 00000000 00000000 00010000 00000000 3c23d70a 00000000 00000000
20020ea0: 00000000 20020f78 00000000 2001ed20 20020fa4 2001f498 2001f1a8 2001f500
20020ec0: 2001f520 00000003 2001f170 ffffffe9 3b831ad2 3c23d70a 00000000 00000000
20020ee0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
20020f00: 00000000 00000000 00000000 00000000 2001f4f0 2001f4a0 3d093964 00000001
20020f20: 00000000 0808ae91 20012d10 2001da40 0000260b 2001f577 2001da40 0000260b
20020f40: 2001f1a8 08087fd7 08087f9d 080cf448 0000260b 080afab1 080afa9d 00000003
20020f60: 2001f577 0809c577 2001ed20 2001f4d8 2001f498 0805e077 2001f568 20024540
20020f80: 00000000 00000000 00000000 0000260b 3d093a57 00000000 2001f540 2001f4f0
20020fa0: 0000260b 3ea5b000 3ddbf5fa 00000000 3c23d70a 00000000 00000000 000f423f
20020fc0: 00000000 000182b8 20017cc0 2001ed20 2001f4e8 00000000 2001f120 0805ea0d
20020fe0: 2001f090 2001f120 2001eda8 ffffffff 000182b8 00000000 00000000 00000000
20021000: 00000000 00000000 00000009 00000000 08090001 2001f93c 0000000c 00000000
20021020: 00000101 2001f96c 00000000 00000000 00000000 00000000 00000000 00000000
20021040: 00000000 00000000 00000000 00000000 00000000 0809866d 00000000 00000000
R0: 20000f48 0a91ae0c 20020d00 20020d00 2001f578 080ca038 000182b8 20017cc0
R8: 2001ed20 2001f4e8 2001ed20 00000005 20020d20 20020ce8 0808439f 08087c4e
xPSR: 61000000 BASEPRI: 00000000 CONTROL: 00000000
EXC_RETURN: ffffffe9

```

To decode the hardfault, load the **exact** binary into the debugger:

```
arm-none-eabi-gdb build_px4fmu-v2_default/src/firmware/nuttx/firmware_nuttx
```

Then in the GDB prompt, start with the last instructions in R8, with the first address in flash (recognizable because it starts with `0x080`, the first is `0x0808439f`). The execution is left to right. So one of the last steps before the hard fault was when `mavlink_log.c` tried to publish something,

```
(gdb) info line *0x0808439f
Line 77 of ".../src/modules/systemlib/mavlink_log.c" starts at address 0x8084398 <mavlink_vasprintf+36>
and ends at 0x80843a0 <mavlink_vasprintf+44>.
```

```
(gdb) info line *0x08087c4e
Line 311 of ".../src/modules/uORB/uORBDevices_nuttx.cpp"
starts at address 0x8087c4e <uORB::DeviceNode::publish(orb_metadata const*, void*,
void const*)+2>
and ends at 0x8087c52 <uORB::DeviceNode::publish(orb_metadata const*, void*, void const*)+6>.
```

Simulation Debugging

As the simulation is running on the host machine, all the desktop development tools are available.

CLANG Address Sanitizer (Mac OS, Linux)

The Clang address sanitizer can help to find alignment (bus) errors and other memory faults like segmentation faults. The command below sets the right compile options.

```
make clean # only required on first address sanitizer run after a normal build  
MEMORY_DEBUG=1 make posix jmavsim
```

Valgrind

```
brew install valgrind
```

or

```
sudo apt-get install valgrind
```

Add instructions how to run Valgrind

Start combinations

SITL can be launched with and without debugger attached and with either jMAVSim or Gazebo as simulation backend. This results in the start options below:

```

make posix_sitl_default jmavsim
make posix_sitl_default jmavsim__gdb
make posix_sitl_default jmavsim__lldb

make posix_sitl_default gazebo
make posix_sitl_default gazebo__gdb
make posix_sitl_default gazebo__lldb

make posix_sitl_lpe jmavsim
make posix_sitl_lpe jmavsim__gdb
make posix_sitl_lpe jmavsim__lldb

make posix_sitl_lpe gazebo
make posix_sitl_lpe gazebo__gdb
make posix_sitl_lpe gazebo__lldb

```

where the last parameter is the <viewer_model_debugger> triplet (using three underscores implies the default 'iris' model). This will start the debugger and launch the SITL application. In order to break into the debugger shell and halt the execution, hit `CTRL-C` :

```

Process 16529 stopped
* thread #1: tid = 0x114e6d, 0x00007fff90f4430a libsystem_kernel.dylib`__read_nocancel
+ 10, name = 'px4', queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
    frame #0: 0x00007fff90f4430a libsystem_kernel.dylib`__read_nocancel + 10
libsystem_kernel.dylib`__read_nocancel:
-> 0x7fff90f4430a <+10>: jae    0x7fff90f44314          ; <+20>
    0x7fff90f4430c <+12>: movq   %rax, %rdi
    0x7fff90f4430f <+15>: jmp    0x7fff90f3fc53          ; cerror_nocancel
    0x7fff90f44314 <+20>: retq
(lldb)

```

In order to not have the DriverFrameworks scheduling interfere with the debugging session `SIGCONT` should be masked in LLDB and GDB:

```
(lldb) process handle SIGCONT -n false -p false -s false
```

Or in the case of GDB:

```
(gdb) handle SIGCONT noprint nostop
```

After that the lldb or gdb shells behave like normal sessions, please refer to the LLDB / GDB documentation.

The last parameter, the <viewer_model_debugger> triplet, is actually passed to make in the build directory, so

```
make posix_sitl_lpe jmavsim__gdb
```

is equivalent with

```
make posix_sitl_lpe      # Configure with cmake
make -C build_posix_sitl_lpe jmavsim__gdb
```

A full list of the available make targets in the build directory can be obtained with:

```
make help
```

but for your convenience, a list with just the <viewer_model_debugger> triplets is printed with the command

```
make list_vmd_make_targets
```

Compiler optimization

It is possible to suppress compiler optimization for given executables and/or modules (as added by cmake with `add_executable` or `add_library`) when configuring for `posix_sitl_*`. This can be handy when it is necessary to step through code with a debugger or print variables that would otherwise be optimized out.

To do so, set the environment variable `PX4_NO_OPTIMIZATION` to be a semi-colon separated list of regular expressions that match the targets that need to be compiled without optimization. This environment variable is ignored when the configuration isn't

```
posix_sitl_* .
```

For example,

```
export PX4_NO_OPTIMIZATION='px4;^modules__uORB;^modules__systemlib$'
```

would suppress optimization of the targets: `platforms__posix_px4_layer`, `modules__systemlib`, `modules__uORB`, `examples_px4_simple_app`, `modules__uORB_uORB_tests` and `px4`.

The targets that can be matched with these regular expressions can be printed with the command:

```
make -C build_posix_sitl_* list_cmake_targets
```

Send Debug String / Float Pairs

It is often necessary during software development to output individual important numbers. This is where the generic `NAMED_VALUE` packets of MAVLink come in.

Files

The code for this tutorial is available here:

- [Debug Tutorial Code](#)
- [Enable the tutorial app](#) by uncommenting / enabling the mavlink debug app in the config of your board

All required to set up a debug publication is this code snippet. First add the header file:

```
#include <uORB/uORB.h>
#include <uORB/topics/debug_key_value.h>
```

Then advertise the debug value topic (one advertisement for different published names is sufficient). Put this in front of your main loop:

```
/* advertise debug value */
struct debug_key_value_s dbg = { .key = "velx", .value = 0.0f };
orb_advert_t pub_dbg = orb_advertise(ORB_ID(debug_key_value), &dbg);
```

And sending in the main loop is even simpler:

```
dbg.value = position[0];
orb_publish(ORB_ID(debug_key_value), pub_dbg, &dbg);
```

Multiple debug messages must have enough time between their respective publishings for Mavlink to process them. This means that either the code must wait between publishing multiple debug messages, or alternate the messages on each function call iteration. The result in QGroundControl then looks like this on the real-time plot:

Sending Debug Values



FAKE GPS

This page shows you how to use mocap data to fake gps.

The setup looks as follows: There is a "VICON computer" which has the required software installed + [ROS](#) + [vicon_bridge](#) and sends this data over network to "your Computer". On "your computer" you should have ROS + MAVROS installed. In MAVROS there is a script which simulates gps data out of mocap data. "Your computer" then sends the data over 3DR radiometry to the pixhawk. *NOTE:* The "VICON computer" and "your computer" can be the same, of course.

Prerequisites

- MOCAP system (in this example, VICON is used)
- Computer with ROS, MAVROS and Vicon_bridge
- 3DR radiometry set

Procedure

Step 1

Make sure "your computer" is in the same network as the "VICON computer" (maybe you need a wireless adapter). Create two files on the "VICON computer": "launch_fake_gps.sh" and "launch_fake_gps_distorted.sh"

Add the following two lines to the "launch_fake_gps.sh" file and replace xxx.xxx.x.xxx with the IP adress of "your computer" (you can get the IP adress by typing "ifconfig" in the terminal).

```
export ROS_MASTER_URI=http://xxx.xxx.x.xxx:11311  
roslaunch vicon_bridge vicon.launch $@
```

Next, add the following two lines to the "launch_fake_gps_distorted.sh" file and replace xxx.xxx.x.xxx with the IP adress of "your computer".

```
export ROS_MASTER_URI=http://xxx.xxx.x.xxx:11311  
rosrun vicon_bridge tf_distort $@
```

Put markers on your drone and create a model in the MOCAP system (later referred to as `yourModelName`).

Step 2

Run

```
$ roscore
```

on "your computer".

Step 3

Run

```
$ sh launch_fake_gps.sh
```

and

```
$ sh launch_fake_gps_distorted.sh
```

in two different terminals on the "VICON computer" in the directory where you created the two files.

Step 4

On "your computer" run

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

A new window should open where you can select "tf_distort". With this tool you can edit the parameters to distort the MOCAP data.

To simulate GPS we use:

- publish rate = 5.0Hz
- tf_frame_in = vicon/`yourModelName`/`yourModelName` (e.g. vicon/DJI_450/DJI_450)
- delay = 200ms
- sigma_xy = 0.05m
- sigma_z = 0.05m

Step 5

Connect your pixhawk in QGroundControl. Go to PARAMETERS -> System and change SYS_COMPANION to 257600 (enables magic mode;)).

Next, go to PARAMETERS -> MAVLink and change MAV_USEHILGPS to 1 (enable HIL GPS).

Now, go to PARAMETERS -> Attitude Q estimator and change ATT_EXT_HDG_M to 2 (use heading from motion capture).

Last but not least, go to PARAMETERS -> Position Estimator INAV and change INAV_DISABLE_MOCAP to 1 (disable mocap estimation).

NOTE: if you can't find the above stated parameters, check PARAMETERS -> default Group

Step 6

Next, open "mocap_fake_gps.cpp". You should find it at:

yourCatkinWS/src/mavros/mavros_extras/src/plugins/mocap_fake_gps.cpp

Replace DJI_450/DJI_450 in

```
mocap_tf_sub = mp_nh.subscribe("/vicon/DJI_450/DJI_450_drop", 1, &MocapFakeGPSPlugin::  
mocap_tf_cb, this);
```

with your model name (e.g. /vicon/yourModelName/yourModelName_drop). The "_drop" will be explained in the next step.

Step 7

In step 5, we enabled heading from motion capture. Therefore pixhawk does not use the original North, East direction, but the one from the motion capture system. Because the 3DR radiometry device is not fast enough, we have to limit the rate of our MOCAP data. To do this run

```
$ rosrun topic_tools drop /vicon/yourModelName/yourModelName 9 10
```

This means, that from the rostopic /vicon/yourModelName/yourModelName, 9 out of 10 messages will be dropped and published under the topic name "/vicon/yourModelName/yourModelName_drop".

Step 8

Connect the 3DR radiometry with the pixhawk TELEM2 and the counter part with your computer (USB).

Step 9

Go to your catkinWS and run

```
$ catkin build
```

and afterwards

```
$ roslaunch mavros px4.launch fcu_url:=/dev/ttyUSB0:57600
```

That's it! Your pixhawk now gets GPS data and the light should pulse in green color.

Camera Trigger

The camera trigger driver allows the use of the AUX ports to send out pulses in order to trigger a camera. This can be used for multiple applications including timestamping photos for aerial surveying and reconstruction, synchronizing a multi-camera system or visual-inertial navigation.

In addition to a pulse being sent out, a MAVLink message is published containing a sequence number (thus the current session's image sequence number) and the corresponding time stamp.

Three different modes are supported:

- `TRIG_MODE` 1 works like a basic intervalometer that can be enabled and disabled by calling in the system console `camera_trigger enable` or `camera_trigger disable`, respectively. Repeated enabling time-shifts the intervals to match the latest call.
- `TRIG_MODE` 2 switches the intervalometer constantly on.
- `TRIG_MODE` 3 triggers based on distance. A shot is taken every time the set horizontal distance is exceeded. The minimum time interval between two shots is however limited by the set triggering interval.

In `TRIG_MODE` 0 the triggering is off.

The full list of parameters pertaining to the camera trigger module can be found on the [parameter reference](#) page.

Info If it is your first time enabling the camera trigger app, remember to reboot after changing the `TRIG_MODE` parameter to either 1, 2 or 3.

Camera-IMU sync example

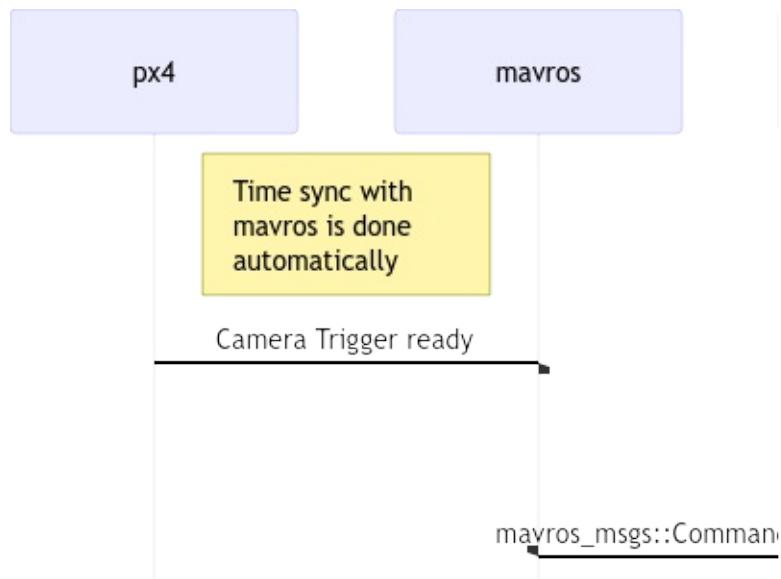
In this example, we will go over the basics of synchronizing IMU measurements with visual data to build a stereo Visual-Inertial Navigation System (VINS). To be clear, the idea here isn't to take an IMU measurement exactly at the same time as we take a picture but rather to correctly time stamp our images so as to provide accurate data to our VI algorithm.

The autopilot and companion have different clock bases (boot-time for the autopilot and UNIX epoch for companion), so instead of skewing either clock, we directly observe the time offset between the clocks. This offset is added or subtracted from the timestamps in the mavlink messages (e.g. `HIGHRES_IMU`) in the cross-middleware translator component (e.g. Mavros on the companion and `mavlink_receiver` in PX4). The actual synchronisation

algorithm is a modified version of the Network Time Protocol (NTP) algorithm and uses an exponential moving average to smooth the tracked time offset. This synchronisation is done automatically if Mavros is used with a high-bandwidth on-board link.

For acquiring synchronised image frames and inertial measurements, we connect the trigger inputs of the two cameras to a GPIO pin on the autopilot. The timestamp of the inertial measurement from mid-exposure, and a image sequence number is recorded and sent to the companion computer (`CAMERA_TRIGGER` message), which buffers these packets and the image frames acquired from the camera. They are matched based on the sequence number, the images timestamped (with the timestamp from the `CAMERA_TRIGGER` message) and then published.

The following diagram illustrates the sequence of events which must happen in order to correctly timestamp our images.



Step 1

First, set the TRIG_MODE to 1 to make the driver wait for the start command and reboot your FCU to obtain the remaining parameters.

Step 2

For the purposes of this example we will be configuring the trigger to operate in conjunction with a Point Grey Firefly MV camera running at 30 FPS.

- TRIG_INTERVAL: 33.33 ms
- TRIG_POLARITY: 0, active low
- TRIG_ACT_TIME: 0.5 ms, leave default. The manual specifies it only has to be a minimum of 1 microsecond.
- TRIG_MODE: 1, because we want our camera driver to be ready to receive images before starting to trigger. This is essential to properly process sequence numbers.
- TRIG_PINS: 12, Leave default.

Step 3

Wire up your cameras to your AUX port by connecting the ground and signal pins to the appropriate place.

Step 4

You will have to modify your driver to follow the sequence diagram above. Public reference implementations for [IDS Imaging UEye](#) cameras and for [IEEE1394 compliant](#) cameras are available.

Logging

This describes the new logger, `SYS_LOGGER` set to 1.

The logger is able to log any ORB topic with all included fields. Everything necessary is generated from the `.msg` file, so that only the topic name needs to be specified. An optional interval parameter specifies the maximum logging rate of a certain topic. All existing instances of a topic are logged.

The output log format is [ULog](#).

Usage

By default, logging is automatically started when arming, and stopped when disarming. A new log file is created for each arming session on the SD card. To display the current state, use `logger status` on the console. If you want to start logging immediately, use `logger on`. This overrides the arming state, as if the system was armed. `logger off` undoes this.

Use

```
logger help
```

for a list of all supported logger commands and parameters.

Configuration

The list of logged topics can be customized with a file on the SD card. Create a file `etc/logging/logger_topics.txt` on the card with a list of topics:

```
<topic_name>, <interval>
```

The `<interval>` is optional, and if specified, defines the minimum interval in ms between two logged messages of this topic. If not specified, the topic is logged at full rate.

The topics in this file replace all of the default logged topics.

Scripts

There are several scripts to analyze and convert logging files in the [pyulog](#) repository.

Dropouts

Logging dropouts are undesired and there are a few factors that influence the amount of dropouts:

- Most SD cards we tested exhibit multiple pauses per minute. This shows itself as a several 100 ms delay during a write command. It causes a dropout if the write buffer fills up during this time. This effect depends on the SD card (see below).
- Formatting an SD card can help to prevent dropouts.
- Increasing the log buffer helps.
- Decrease the logging rate of selected topics or remove unneeded topics from being logged (`info.py <file>` is useful for this).

SD Cards

The following provides performance results for different SD cards. Tests were done on a Pixracer; the results are applicable to Pixhawk as well.

SD Card	Mean Seq. Write Speed [KB/s]	Max Write Time / Block (average) [ms]
SanDisk Extreme U3 32GB	461	15
Sandisk Ultra Class 10 8GB	348	40
Sandisk Class 4 8GB	212	60
SanDisk Class 10 32 GB (High Endurance Video Monitoring Card)	331	220
Lexar U1 (Class 10), 16GB High-Performance	209	150
Sandisk Ultra PLUS Class 10 16GB	196	500
Sandisk Pixtor Class 10 16GB	334	250
Sandisk Extreme PLUS Class 10 32GB	332	150

More important than the mean write speed is the maximum write time per block (of 4 KB). This defines the minimum buffer size: the larger this maximum, the larger the log buffer needs to be to avoid dropouts. Logging bandwidth with the default topics is around 50 KB/s, which all of the SD cards satisfy.

By far the best card we know so far is the **SanDisk Extreme U3 32GB**. This card is recommended, because it does not exhibit write time spikes (and thus virtually no dropouts). Different card sizes might work equally well, but the performance is usually different.

You can test your own SD card with `sd_bench -r 50`, and report the results to <https://github.com/PX4/Firmware/issues/4634>.

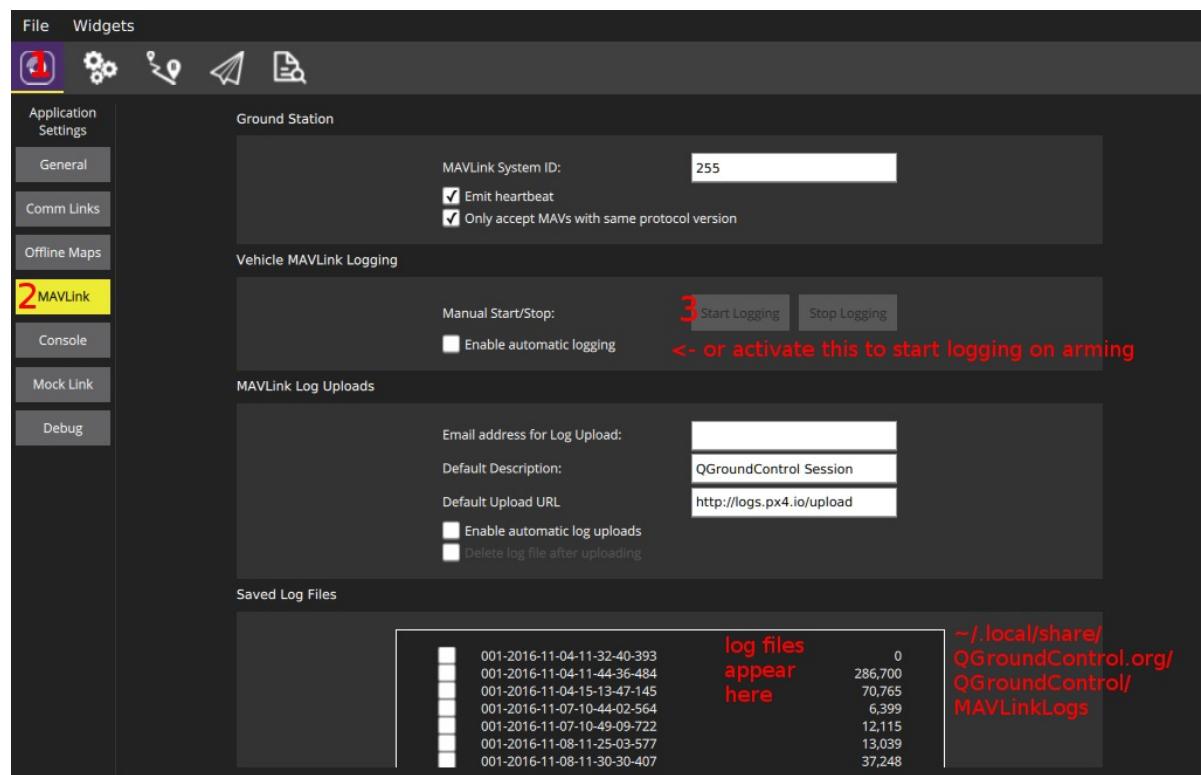
Log Streaming

The traditional and still fully supported way to do logging is using an SD card on the FMU. However there is an alternative, log streaming, which sends the same logging data via MAVLink. This method can be used for example in cases where the FMU does not have an SD card slot (eg. Intel Aero) or simply to avoid having to deal with SD cards. Both methods can be used independently and at the same time.

The requirement is that the link provides at least ~50KB/s, so for example a WiFi link. And only one client can request log streaming at the same time. The connection does not need to be reliable, the protocol is designed to handle drops.

There are different clients that support uLog streaming:

- `mavlink_ugc_streaming.py` script in Firmware/Tools.
- QGroundControl:



- MAVGCL

Diagnostics

- If log streaming does not start, make sure the `logger` is running (see above), and inspect the console output while starting.
- Log streaming uses a maximum of 70% of the configured mavlink rate (`-r` parameter). If more is needed, messages are dropped. The currently used percentage can be inspected with `mavlink status` (1.8% is used in this example):

```
instance #0:  
    GCS heartbeat: 160955 us ago  
    mavlink chan: #0  
    type:           GENERIC LINK OR RADIO  
    flow control:  OFF  
    rates:  
        tx: 95.781 kB/s  
        txerr: 0.000 kB/s  
        rx: 0.021 kB/s  
        rate mult: 1.000  
    ULog rate: 1.8% of max 70.0%  
    accepting commands: YES  
    MAVLink version: 2  
    transport protocol: UDP (14556)
```

Also make sure `txerr` stays at 0. If this goes up, either the NuttX sending buffer is too small, the physical link is saturated or the hardware is too slow to handle the data.

Flight Log Analysis

There are several software packages that exist to analyze PX4 flight logs. They are described below.

Log Muncher

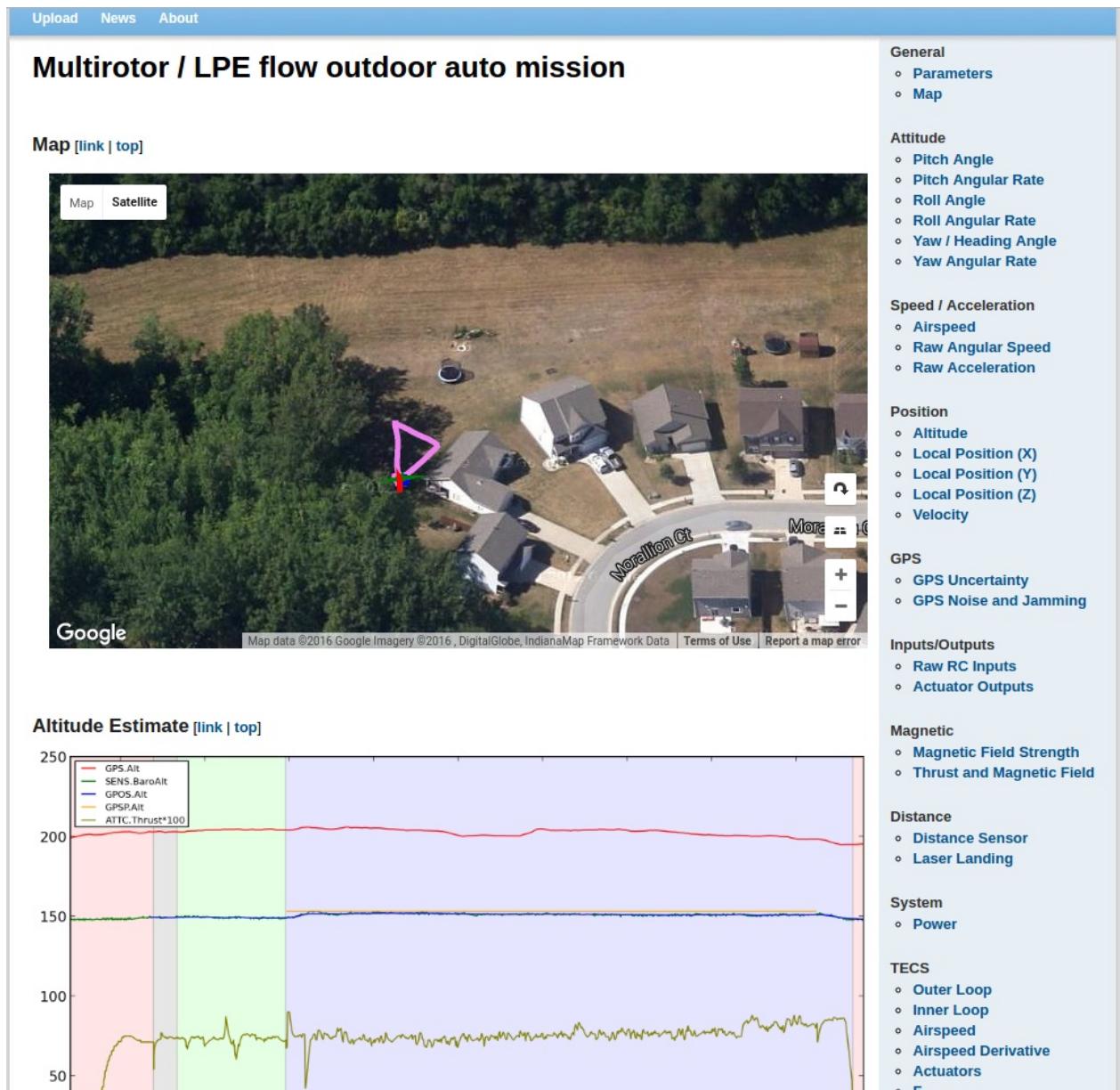
Note: Log Muncher can only be used with the previous log format (.px4log).

Upload

Users can visit this webpage and upload log files directly: <http://logs.uaventure.com/>

The screenshot shows the Log Muncher website interface. At the top, there is a dark header bar with the text "LogMuncher [Flight Log Analyser]". Below it is a light blue navigation bar with links for "Upload", "News", and "About". The main content area has a white background. On the left, there is a form titled "Upload" with fields for "Email" (an input field), "Title" (another input field), and a file upload section with a "Choose File" button and a message "No file chosen". To the right of this form is a vertical sidebar with a light blue background and a white border. It contains a section titled "Important" which contains a detailed note about uploaded log files. At the bottom of the sidebar, there is a copyright notice: "Copyright © UAVenture AG".

Result



Example Log

Positive

- web based, great for end-users
- user can upload, load and then share report with others

Negative

- analysis is very constrained, no customization possible

Flight Review

Flight Review is the successor of Log Muncher, used in combination with the new ULog logging format.

Example



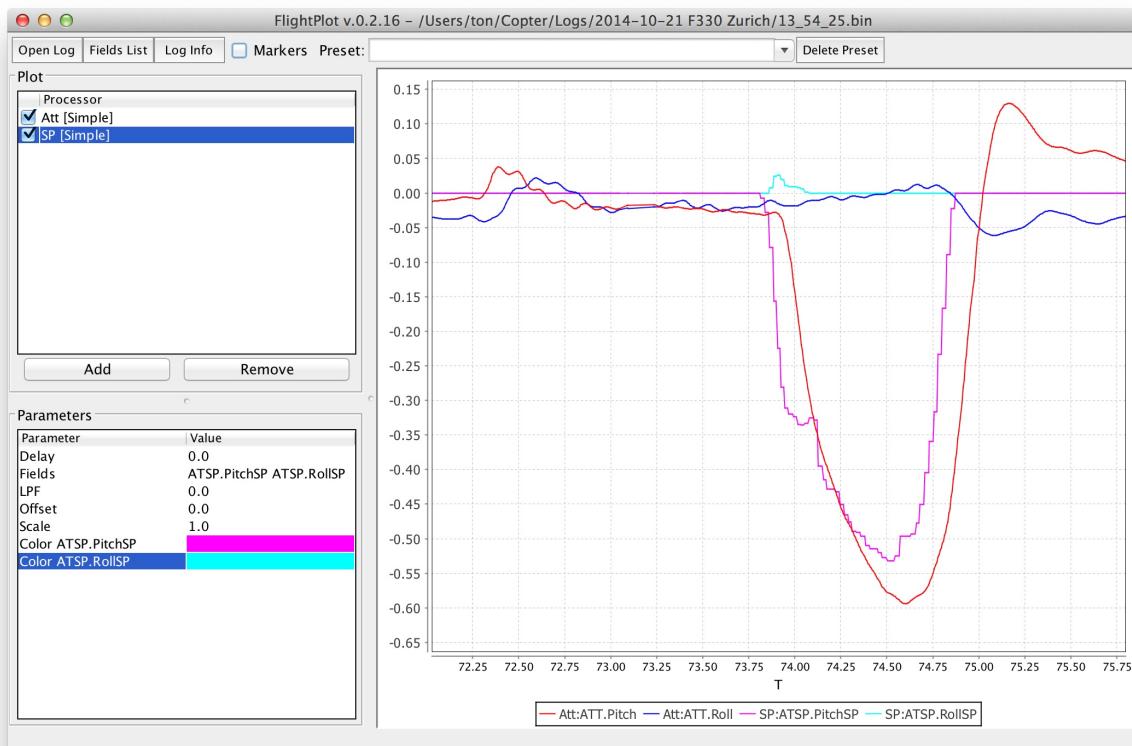
Positive

- web based, great for end-users
- user can upload, load and then share report with others
- interactive plots

Negative

- no customization possible

FlightPlot



- [FlightPlot Downloads](#)

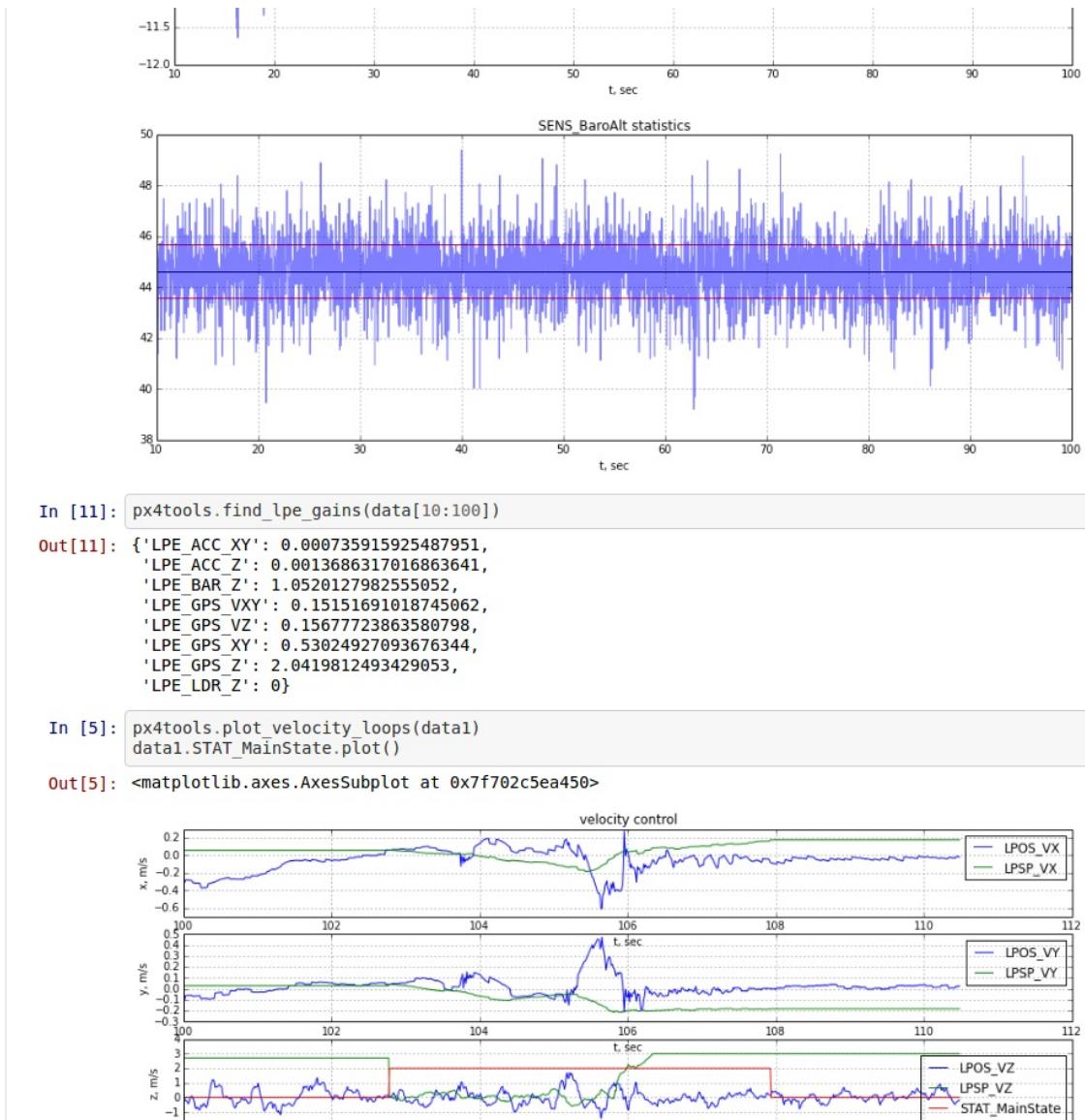
Positive

- java based, cross-platform
- intuitive GUI, no programming knowledge required

Negative

- analysis constrained by what features have been built-in

PX4Tools



Install

- The recommended procedure is to use anaconda3. See [px4tools github page](#) for details.

```
conda install -c https://conda.anaconda.org/dronecrew px4tools
```

Positive

- easy to share, users can view notebooks on github (e.g. <https://github.com/jgoppert/lpe-analysis/blob/master/15-09-30%20Kabir%20Log.ipynb>)
- python based, cross platform, works with anaconda 2 and anaconda3
- ipython/ jupyter notebooks can be used to share analysis easily
- advanced plotting capabilities to allow detailed analysis

Negative

- Requires the user to know python
- Currently requires use of sdlog2_dump.py or px4tools embedded px42csv program to convert log files to csv before use

EKF2 Log Replay

This page shows you how you can tune the parameters of the EKF2 estimator by using the replay feature on a real flight log. It is based on the `sdlog2` logger.

Introduction

A developer has the possibility to enable onboard logging of the EKF2 estimator input sensor data. This allows him to tune the parameters of the estimator offline by running a replay on the logged data trying out different sets of parameters. The remainder of this page will explain which parameters have to be set in order to benefit from this feature and how to correctly deploy it.

Prerequisites

- set the parameter **EKF2_REC_RPL** to 1. This will tell the estimator to publish special replay messages for logging.
- set the parameter **SDLOG_PRIO_BOOST** to a value contained in the set {0, 1, 2, 3}. A value of 0 means that the onboard logging app has a default (low) scheduling priority. A low scheduling priority can lead to a loss of logging messages. If you find that your log file contains 'gaps' due to skipped messages then you can increase this parameter to a maximum value of 3. Testing has shown that a minimum value of 2 is required in order to avoid loss of data.

Deployment

Once you have a real flight log then you can run a replay on it by using the following command in the root directory of your PX4 Firmware

```
make posix_sitl_replay replay logfile=absolute_path_to_log_file/my_log_file.px4log
```

where 'absolute_path_to_log_file/my_log_file.px4log' is a placeholder for the absolute path of the log file you want to run the replay on. Once the command has executed check the terminal for the location and name of the replayed log file.

Changing tuning parameters for a replay

You can set the estimator parameter values for the replay in the file **replay_params.txt** located in the same directory as your replayed log file, e.g.

build_posix_sitl_replay/src/firmware posix/rootfs/replay_params.txt. When running the replay for the first time (e.g. after a **make clean**) this file will be autogenerated and filled with the default EKF2 parameter values taken from the flight log. After that you can change any EKF2 parameter value by changing the corresponding number in the text file. Setting the noise value for the gyro bias would require the following line.

```
EKF2_GYRO_BIAS_NOISE 0.001
```

Installing driver on Ubuntu for Intel RealSense R200

This tutorial aims to give instructions on how to install the camera driver of the Intel RealSense R200 camera head in Linux environment such that the gathered images can be accessed via the Robot Operation System (ROS). The RealSense R200 camera head is depicted below:



The installation of the driver package is executed on a Ubuntu operation system (OS) that runs as a guest OS in a Virtual Box. The specifications of the host computer where the Virtual Box is running, the Virtual Box and the guest system are given below:

- Host Operation System: Windows 8
- Processor: Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz
- Virtual Box: Oracle VM. Version 5.0.14 r105127
- Extensions: Extension package for Virtual Box installed (Needed for USB3 support)
- Guest Operation System: Linux - Ubuntu 14.04.3 LTS

The tutorial is ordered in the following way: In a first part it is shown how to install Ubuntu 14.04 as a guest OS in the Virtual Box. In a second part is shown how to install ROS Indigo and the camera driver. The ensuing frequently used expressions have the following meaning:

- Virtual Box (VB): Program that runs different Virtual Machines. In this case the Oracle VM.
- Virtual Machine (VM): The operation system that runs in the Virtual Box as a guest system. In this case Ubuntu.

Installing Ubuntu 14.04.3 LTS in Virtual Box

- Create a new Virtual Machine (VM): Linux 64-Bit.
- Download the iso file of Ubuntu 14.04.3 LTS: ([ubuntu-14.04.3-desktop-amd64.iso](#)).
- Installation of Ubuntu:
 - During the installation procedure leave the following two options unchecked:
 - Download updates while installing
 - Install this third party software
- After the installation you might need to enable the Virtual Box to display Ubuntu on the whole desktop:
 - Start VM Ubuntu and login, Click on Devices->Insert Guest Additions CD image in the menu bar of the Virtual Box.
 - Click on "Run" and enter password on the windows that pop up in Ubuntu.
 - Wait until the installation is completed and then restart. Now, it should be possible to display the VM on the whole desktop.
 - If a window pops up in Ubuntu that asks whether to update, reject to update at this point.
- Enable USB 3 Controller in Virtual Box:
 - Shut down Virtual Machine.
 - Go to the settings of the Virtual Machine to the menu selection USB and choose: "USB 3.0(xHCI)". This is only possible if you have installed the extension package for the Virtual Box.
 - Start the Virtual Machine again.

Installing ROS Indigo

- Follow instructions given at [ROS indigo installation guide](#):
 - Install Desktop-Full version.
 - Execute steps described in the sections "Initialize rosdep" and "Environment setup".

Installing camera driver

- Install git:

```
sudo apt-get install git
```
- Download and install the driver
 - Clone [RealSense_ROS repository](#):

```
git clone https://github.com/PercATI/RealSense_ROS.git
```

- Follow instructions given in [here](#).
 - Press the enter button when the questions whether to install the following installation packages show up:

```
Intel Low Power Subsystem support in ACPI mode (MFD_INTEL_LPSS_ACPI) [N/m/y/?] (NEW)
```

```
Intel Low Power Subsystem support in PCI mode (MFD_INTEL_LPSS_PCI) [N/m/y/?] (NEW)
```

```
Dell Airplane Mode Switch driver (DELL_RBTN) [N/m/y/?] (NEW)
```

- The following error message that can appear at the end of the installation process should not lead to a malfunction of the driver:

```
rmmmod: ERROR: Module uvcvideo is not currently loaded
```

- After the installation has completed, reboot the Virtual Machine.
- Test camera driver:
 - Connect the Intel RealSense camera head with the computer with a USB3 cable that is plugged into a USB3 receptacle on the computer.
 - Click on Devices->USB-> Intel Corp Intel RealSense 3D Camera R200 in the menu bar of the Virtual Box, in order to forward the camera USB connection to the Virtual Machine.
 - Execute the file [unpacked folder]/Bin/DSReadCameraInfo:
 - If the following error message appears, unplug the camera (physically unplug USB cable from the computer). Plug it in again + Click on Devices->USB-> Intel Corp Intel RealSense 3D Camera R200 in the menu bar of the Virtual Box again and execute again the file [unpacked folder]/Bin/DSReadCameraInfo.
- DSAPI call failed at ReadCameraInfo.cpp:134!
- If the camera driver works and recognises the Intel RealSense R200, you should see specific information about the Intel RealSense R200 camera head.
- Installation and testing of the ROS nodlet:

- Follow the installation instructions in the "Installation" section given [here](#), to install the ROS nodlet.
- Follow the instructions in the "Running the R200 nodelet" section given [here](#), to test the ROS nodlet together with the Intel RealSense R200 camera head.
 - If everything works, the different data streams from the Intel RealSense R200 camera are published as ROS topics.

Bebop 2 - Advanced

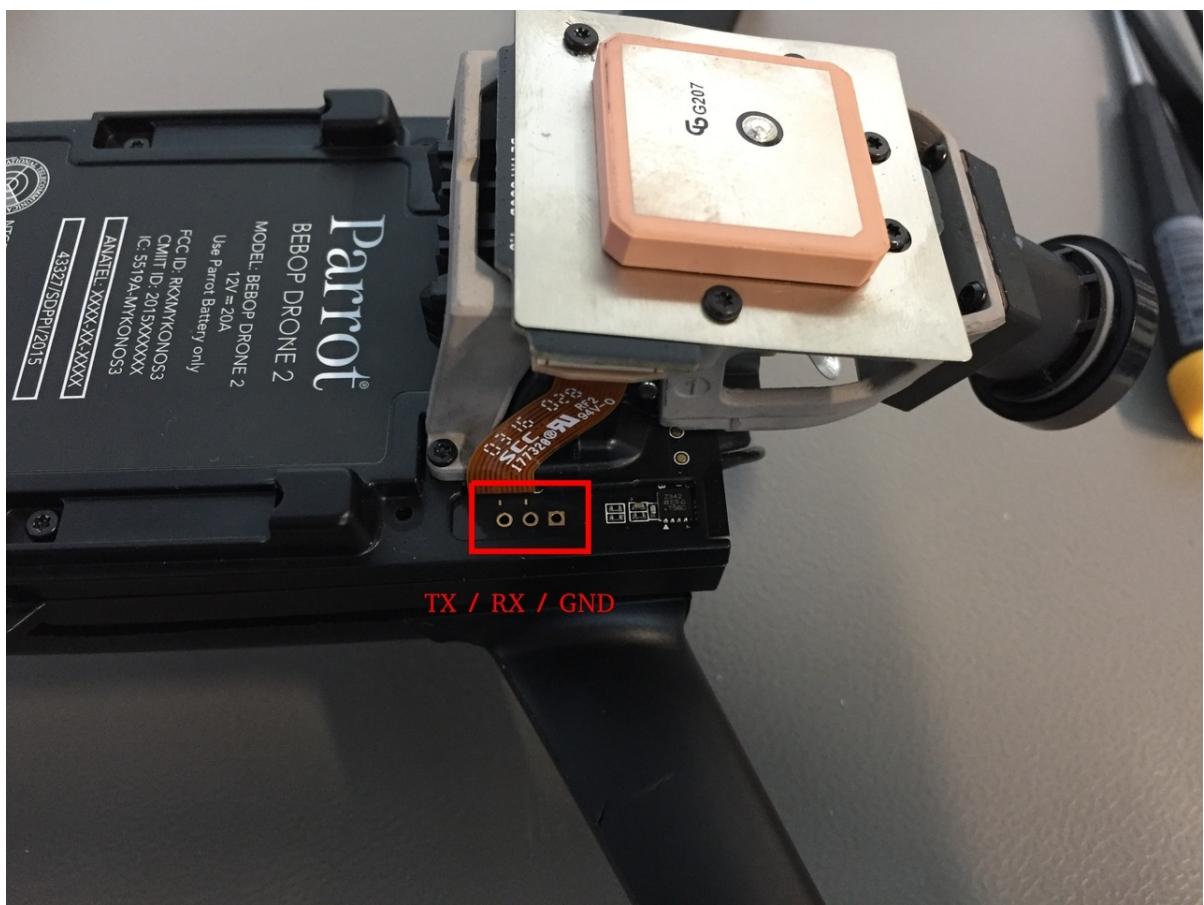
FTDI connection

Follow the instructions to connect to the Parrot Bebop 2 via FTDI.

- Loosen the two Torx screws (T5) to take off the front cap.



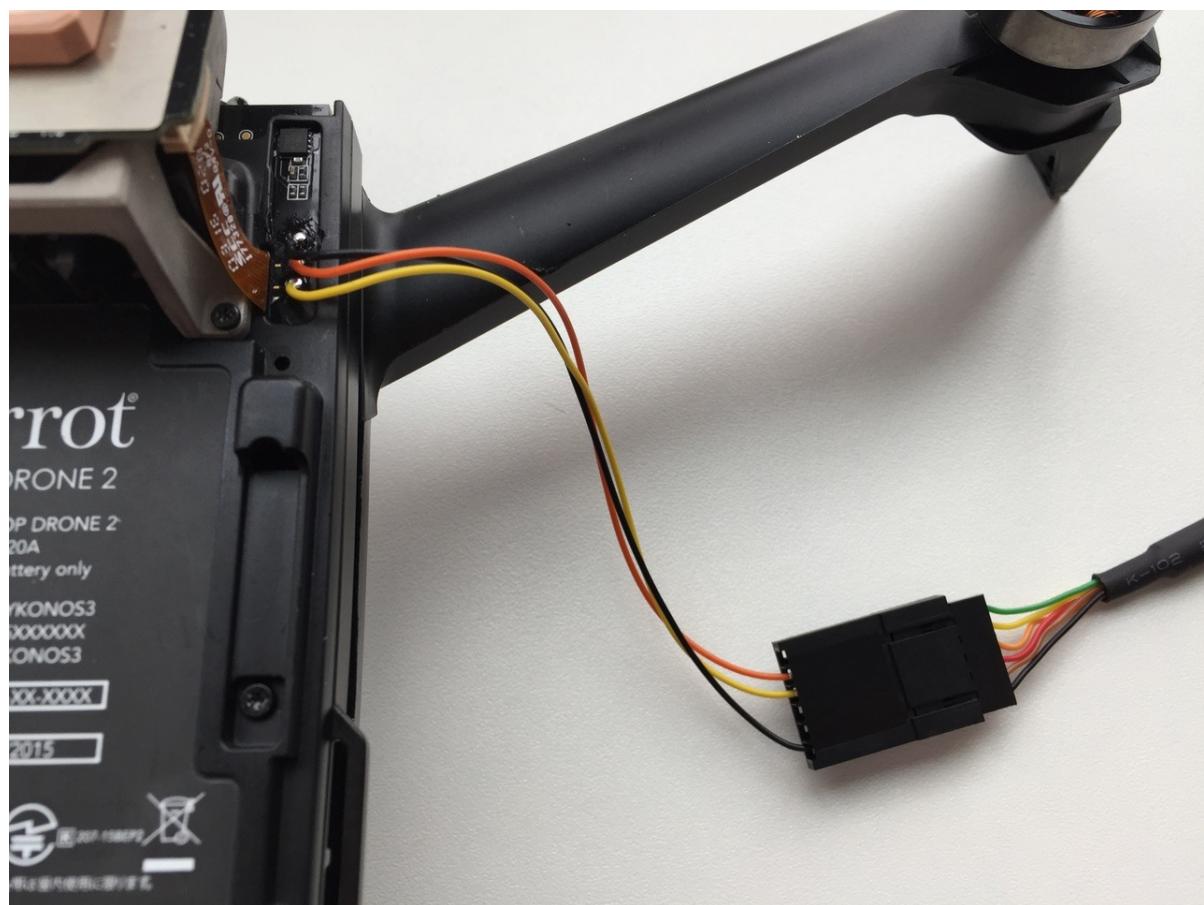
- Use pins to connect to ground/RX/TX or solder cables onto the connectors.



- Connect the FTDI cable and run

```
screen /dev/ttyUSB0 115200
```

to connect to the Bebop.



Gimbal Control Setup

PX4 contains a generic mount/gimbal control driver with different input and output methods. Any input can be selected to drive any output.

First, make sure the driver runs, using `vmount start`, then configure its parameters.

Parameters

The parameters are described in [src/drivers/vmount/vmount_params.c](#). The most important ones are the input (`MNT_MODE_IN`) and the output (`MNT_MODE_OUT`) mode. By default, the input is disabled. Any input method can be selected to drive any of the available outputs.

If a mavlink input mode is selected, manual RC input can be enabled in addition (`MNT_MAN_CONTROL`). It is active as long as no mavlink message is received yet, or mavlink explicitly requests RC mode.

Configure the gimbal mixer for AUX output

The gimbal uses the control group #2 (see [Mixing and Actuators](#)). This is the mixer configuration:

```
# roll
M: 1
O:      10000  10000      0 -10000  10000
S: 2 0  10000  10000      0 -10000  10000

# pitch
M: 1
O:      10000  10000      0 -10000  10000
S: 2 1  10000  10000      0 -10000  10000

# yaw
M: 1
O:      10000  10000      0 -10000  10000
S: 2 2  10000  10000      0 -10000  10000
```

Add those you need to your main or auxiliary mixer.

Testing

The driver provides a simple test command - it needs to be stopped first with `vmount stop`. The following describes testing in SITL, but the commands also work on a real device.

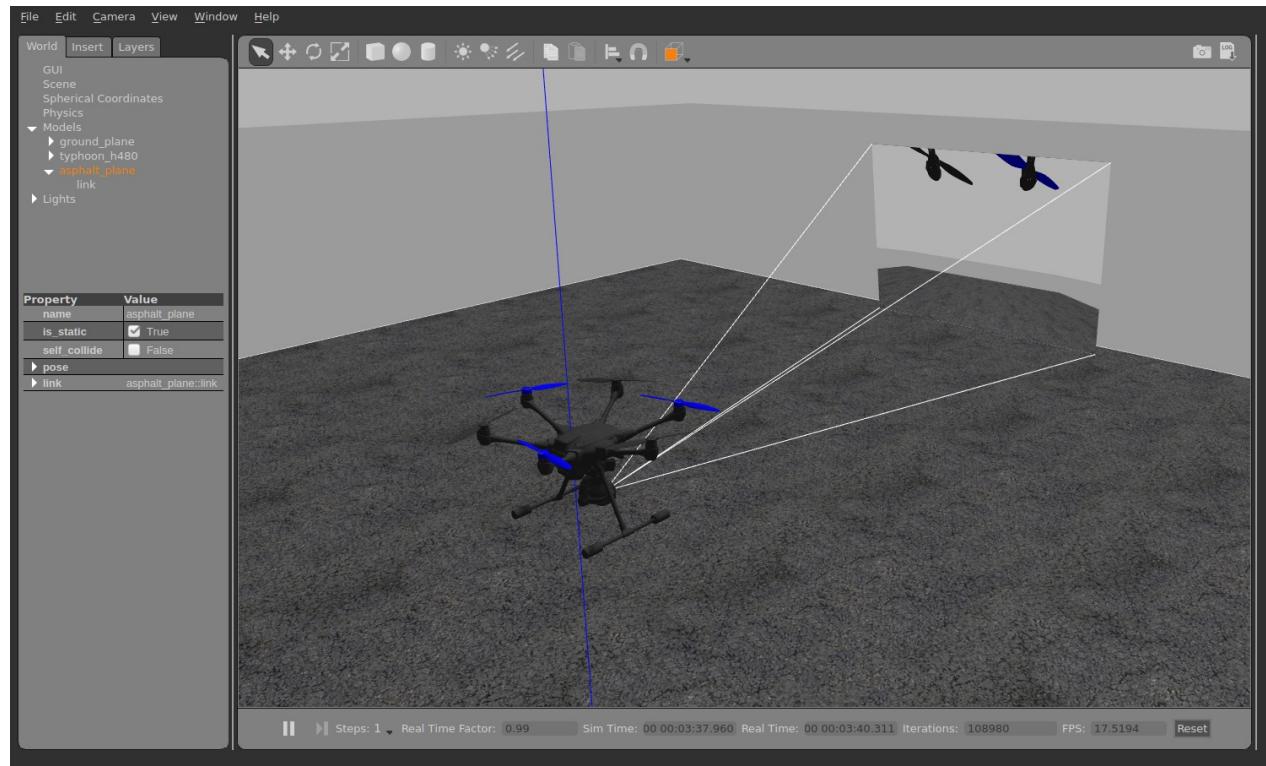
Start the simulation with (no parameter needs to be changed for that):

```
make posix gazebo_typhoon_h480
```

Make sure it's armed, eg. with `commander takeoff`, then use for example

```
vmount test yaw 30
```

to control the gimbal. Note that the simulated gimbal stabilizes itself, so if you send mavlink commands, set the `stabilize` flags to false.



Switching State Estimators

This page shows you which state estimators are available and how you can switch between them.

Available estimators

1. Q attitude estimator

The attitude Q estimator is a very simple, quaternion based complementary filter for attitude.

2. INAV position estimator

The INAV position estimator is a complementary filter for 3D position and velocity states.

3. LPE position estimator

The LPE position estimator is an extended kalman filter for 3D position and velocity states.

4. EKF2 attitude, position and wind states estimator

EKF2 is an extended kalman filter estimating attitude, 3D position / velocity and wind states.

4. EKF attitude, position and wind states estimator (depricated) This is an extened kalman filter similar to EKF2. However, it will soon be replaced completely by EKF2. This filter was only used for fixed wings.

How to enable different estimators

For multirotors and VTOL use the parameter **SYS_MC_EST_GROUP** to chose between the following configurations.

At the moment only the depriciated EKF estimator is used for planes (non VTOL). It will soon be replaced by EKF2.

SYS_MC_EST_GROUP	Q Estimator	INAV	LPE	EKF2
0	enabled	enabled		
1	enabled		enabled	
2				enabled

Out-of-tree Modules

This describes the possibility to add an external module to the PX4 build.

External modules can use the same includes as internal modules and can interact with internal modules via uORB.

Usage

- `EXTERNAL_MODULES_LOCATION` needs to point to a directory with the same structure as Firmware (and thus contains a directory called `src`).
- There are two options: copy an existing module (eg. `examples/px4_simple_app`) to the external directory, or directly create a new module.
- Rename the module (including `MODULE` in `CMakeLists.txt`) or remove it from the existing Firmware `cmake` build config. This is to avoid conflicts with internal modules.
- Add a file `$EXTERNAL_MODULES_LOCATION/CMakeLists.txt` with content:

```
set(config_module_list_external
    modules/<new_module>
    PARENT_SCOPE
)
```

- add a line `EXTERNAL` to the `modules/<new_module>/CMakeLists.txt` within `px4_add_module`, for example like this:

```
px4_add_module(
    MODULE modules__test_app
    MAIN test_app
    STACK_MAIN 2000
    SRCS
        px4_simple_app.c
    DEPENDS
        platforms__common
    EXTERNAL
)
```

- Execute `make posix EXTERNAL_MODULES_LOCATION=<path>`. Any other build target can be used, but the build directory must not yet exist. If it already exists, you can also just set the `cmake` variable in the build folder. For the following incremental builds `EXTERNAL_MODULES_LOCATION` does not need to be specified anymore.

ULog File Format

ULog is the file format used for logging system data. The format is self-describing, i.e. it contains the format and message types that are logged.

It can be used for logging device inputs (sensors, etc.), internal states (cpu load, attitude, etc.) and printf log messages.

The format uses Little Endian for all binary types.

Data types

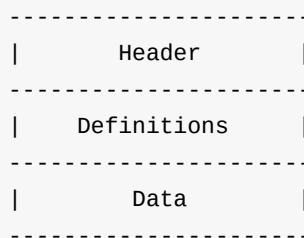
The following binary types are used. They all correspond to the types in C:

Type	Size in Bytes
int8_t, uint8_t	1
int16_t, uint16_t	2
int32_t, uint32_t	4
int64_t, uint64_t	8
float	4
double	8
bool, char	1

Additionally all can be used as an array, eg. `float[5]`. In general all strings (`char[length]`) do not contain a `'\0'` at the end. String comparisons are case sensitive.

File structure

The file consists of three sections:



Header Section

The header is a fixed-size section and has the following format (16 bytes):

```
-----| 0x55 0x4c 0x6f 0x67 0x01 0x12 0x35 | 0x00      | uint64_t      |
| File magic (7B)                   | Version (1B) | Timestamp (8B) |
-----
```

Version is the file format version, currently 0. Timestamp is an `uint64_t` integer, denotes the start of the logging in microseconds.

Definitions Section

Variable length section, contains version information, format definitions, and (initial) parameter values.

The Definitions and Data sections consist of a stream of messages. Each starts with this header:

```
struct message_header_s {
    uint16_t msg_size;
    uint8_t msg_type
};
```

`msg_size` is the size of the message in bytes without the header (`hdr_size = 3 bytes`).

`msg_type` defines the content and is one of the following:

- 'F': format definition for a single (composite) type that can be logged or used in another definition as a nested type.

```
struct message_format_s {
    struct message_header_s header;
    char format[header.msg_size-hdr_size];
};
```

`format` : plain-text string with the following format: `message_name:field0;field1;` There can be an arbitrary amount of fields (at least 1), separated by `;`. A field has the format: `type field_name` or `type[array_length] field_name` for arrays (only fixed size arrays are supported). `type` is one of the basic binary types or a `message_name` of another format definition (nested usage). A type can be used before it's defined. There can be arbitrary nesting but no circular dependencies.

Some field names are special:

- `timestamp` : every logged message (`message_add_logged_s`) must include a timestamp field (does not need to be the first field). Its type can be: `uint64_t` (currently the only one used), `uint32_t`, `uint16_t` or `uint8_t`. The unit is always microseconds, except for `uint8_t` it's milliseconds. A log writer must make sure to log messages often enough to be able to detect wrap-arounds and a log reader must handle wrap-arounds (and take into account dropouts). The timestamp must always be monotonic increasing for a message serie with the same `msg_id`.
- Padding: field names that start with `_padding` should not be displayed and their data must be ignored by a reader. These fields can be inserted by a writer to ensure correct alignment.

If the padding field is the last field, then this field will not be logged, to avoid writing unnecessary data. This means the `message_data_s.data` will be shorter by the size of the padding. However the padding is still needed when the message is used in a nested definition.

- 'I': information message.

```
struct message_info_s {
    struct message_header_s header;
    uint8_t key_len;
    char key[key_len];
    char value[header.msg_size-hdr_size-1-key_len]
};
```

`key` is a plain string, as in the format message, but consists of only a single field without ending `;`, eg. `float[3] myvalues`. `value` contains the data as described by `key`.

Predefined information messages are:

key	Description	Example for value
char[value_len] sys_name	Name of the system	"PX4"
char[value_len] ver_hw	Hardware version	"PX4FMU_V4"
char[value_len] ver_sw	Software version (git tag)	"7f65e01"
uint32_t ver_sw_release	Software version (see below)	0x010401ff
char[value_len] sys_os_name	Operating System Name	"Linux"
char[value_len] sys_os_ver	OS version (git tag)	"9f82919"
uint32_t ver_os_release	OS version (see below)	0x010401ff
char[value_len] sys_toolchain	Toolchain Name	"GNU GCC"
char[value_len] sys_toolchain_ver	Toolchain Version	"6.2.1"
char[value_len] sys_mcu	Chip name and revision	"STM32F42x, rev A"
char[value_len] sys_uuid	Unique identifier for vehicle (eg. MCU ID)	"392a93e32fa3"...
char[value_len] replay	File name of replayed log if in replay mode	"log001.ulg"
int32_t time_ref_utc	UTC Time offset in seconds	-3600

The format of `ver_sw_release` and `ver_os_release` is: 0xAABBCCTT, where AA is major, BB is minor, CC is patch and TT is the type. Type is defined as following: `>= 0` : development, `>= 64` : alpha version, `>= 128` : beta version, `>= 192` : RC version, `== 255` : release version. So for example 0x010402ff translates into the release version v1.4.2.

- 'P': parameter message. Same format as `message_info_s`. If a parameter dynamically changes during runtime, this message can also be used in the Data section. The data type is restricted to: `int32_t`, `float`.

This section ends before the start of the first `message_add_logged_s` or `message_logging_s` message, whichever comes first.

Data Section

The following messages belong to this section:

- 'A': subscribe a message by name and give it an id that is used in `message_data_s`. This must come before the first corresponding `message_data_s`.

```
struct message_add_logged_s {
    struct message_header_s header;
    uint8_t multi_id;
    uint16_t msg_id;
    char message_name[header.msg_size-hdr_size-3];
};
```

`multi_id` : the same message format can have multiple instances, for example if the system has two sensors of the same type. The default and first instance must be 0.
`msg_id` : unique id to match `message_data_s` data. The first use must set this to 0, then increase it. The same `msg_id` must not be used twice for different subscriptions, not even after unsubscribing. `message_name` : message name to subscribe to. Must match one of the `message_format_s` definitions.

- 'R': unsubscribe a message, to mark that it will not be logged anymore (not used currently).

```
struct message_remove_logged_s {
    struct message_header_s header;
    uint16_t msg_id;
};
```

- 'D': contains logged data.

```
struct message_data_s {
    struct message_header_s header;
    uint16_t msg_id;
    uint8_t data[header.msg_size-hdr_size];
};
```

`msg_id` : as defined by a `message_add_logged_s` message. `data` contains the logged binary message as defined by `message_format_s`. See above for special treatment of padding fields.

- 'L': Logged string message, i.e. printf output.

```
struct message_logging_s {
    struct message_header_s header;
    uint8_t log_level;
    uint64_t timestamp;
    char message[header.msg_size-hdr_size-9]
};
```

`timestamp` : in microseconds, `log_level` : same as in the Linux kernel:

Name	Level value	Meaning
EMERG	'0'	System is unusable
ALERT	'1'	Action must be taken immediately
CRIT	'2'	Critical conditions
ERR	'3'	Error conditions
WARNING	'4'	Warning conditions
NOTICE	'5'	Normal but significant condition
INFO	'6'	Informational
DEBUG	'7'	Debug-level messages

- 'S': synchronization message so that a reader can recover from a corrupt message by searching for the next sync message (not used currently).

```
struct message_sync_s {
    struct message_header_s header;
    uint8_t sync_magic[8];
};
```

sync_magic : to be defined.

- 'O': mark a dropout (lost logging messages) of a given duration in ms. Dropouts can occur e.g. if the device is not fast enough.

```
struct message_dropout_s {
    struct message_header_s header;
    uint16_t duration;
};
```

Licenses

This page documents the licenses of various components in the system.

- [PX4 Flight Stack](#) — BSD
- [PX4 Middleware](#) — BSD
- [Pixhawk Hardware](#) — CC-BY-SA 3.0
- Snapdragon Hardware — Qualcomm Proprietary

Software Update

The method to update the PX4 software on the drone depends on the hardware platform. For microcontroller based applications new Firmware is flashed through USB or serial.

Infrastructure

STM32 Bootloader

The code for the PX4 bootloader is available from the Github [Bootloader](#) repository.

Supported Boards

- FMUv1 (PX4FMU, STM32F4)
- FMUv2 (Pixhawk 1, STM32F4)
- FMUv3 (Pixhawk 2, STM32F4)
- FMUv4 (Pixracer 3 and Pixhawk 3 Pro, STM32F4)
- FMUv5 (Pixhawk 4, STM32F7)
- TAPv1 (TBA, STM32F4)
- ASCv1 (TBA, STM32F4)

Building the Bootloader

```
git clone https://github.com/PX4/Bootloader.git  
cd Bootloader  
make
```

After this step a range of elf files for all supported boards are present in the Bootloader directory.

Flashing the Bootloader

IMPORTANT: The right power sequence is critical for some boards to allow JTAG / SWD access. Follow these steps exactly as described. The instructions below are valid for a Blackmagic / Dronecode probe. Other JTAG probes will need different but similar steps. Developers attempting to flash the bootloader should have the required knowledge. If you do not know how to do this you probably should reconsider if you really need to change anything about the bootloader.

- Disconnect the JTAG cable
- Connect the USB power cable
- Connect the JTAG cable

Black Magic / Dronecode Probe

Using the right serial port

- On LINUX: `/dev/serial/by-id/usb-Black_Sphere_XXX-if00`
- On MAC OS: Make sure to use the cu.xxx port, not the tty.xxx port: `tar ext /dev/tty.usbmodemDDEasdf`

```
arm-none-eabi-gdb
(gdb) tar ext /dev/serial/by-id/usb-Black_Sphere_XXX-if00
(gdb) mon swdp_scan
(gdb) attach 1
(gdb) mon option erase
(gdb) mon erase_mass
(gdb) load tapv1_bl.elf
...
Transfer rate: 17 KB/sec, 828 bytes/write.
(gdb) kill
```

J-Link

These instructions are for the [J-Link GDB server](#).

Prerequisites

Download the J-Link software from the Segger website and install it according to their instructions.

Run the JLink GDB server

FMUv1:

```
JLinkGDBServer -select USB=0 -device STM32F405RG -if SWD-DP -speed 20000
```

AeroFC:

```
JLinkGDBServer -select USB=0 -device STM32F429AI -if SWD-DP -speed 20000
```

Connect GDB

```
arm-none-eabi-gdb
(gdb) tar ext :2331
(gdb) load aerofcv1_bl.elf
```

Troubleshooting

If any of the commands above are not found, you are either not using a Blackmagic probe or its software is outdated. Upgrade the on-probe software first.

If this error message occurs: Error erasing flash with vFlashErase packet

Disconnect the target (while leaving JTAG connected) and run

```
mon tpwr disable
swdp_scan
attach 1
load tapv1_bl.elf
```

This will disable target powering and attempt another flash cycle.

Testing and Continuous Integration

PX4 offers extensive unit testing and continuous integration facilities. This page provides an overview.

Testing on the local machine

The following command is sufficient to start a minimal new shell with the PX4 posix port running.

```
make posix_sitl_shell none
```

The shell can then be used to e.g. execute unit tests:

```
pxh> tests mixer
```

Alternatively it is also possible to run the complete unit-tests right from bash:

```
make tests
```

Testing in the Cloud / CI

PX4 Docker Containers

Docker containers are available that contain the complete PX4 development toolchain including Gazebo and ROS simulation:

- **px4io/px4-dev**: toolchain including simulation
- **px4io/px4-dev-ros**: toolchain including simulation and ROS (incl. MAVROS)

Pull one of the tagged images if you're after a container that just works, for instance

`px4io/px4-dev-ros:v1.0`, the `latest` container is usually changing a lot.

Dockerfiles and README can be found here:

<https://github.com/PX4/containers/tree/master/docker/px4-dev>

They are build automatically on Docker Hub: <https://hub.docker.com/u/px4io/>

Prerequisites

Install Docker from here <https://docs.docker.com/installation/>, preferably use one of the Docker-maintained package repositories to get the latest version.

Containers are currently only supported on Linux. If you don't have Linux you can run the container inside a virtual machine, see further down for more information. Do not use `boot2docker` with the default Linux image because it contains no X-Server.

Use the Docker container

The following will run the Docker container including support for X forwarding which makes the simulation GUI available from inside the container. It also maps the directory `<local_src>` from your computer to `<container_src>` inside the container and forwards the UDP port needed to connect QGC. Please see the Docker docs for more information on volume and network port mapping.

With the `--privileged` option it will automatically have access to the devices on your host (e.g. a joystick and GPU). If you connect/disconnect a device you have to restart the container.

```
# enable access to xhost from the container
xhost +

docker run -it --privileged \
-v <local_src>:<container_src>:rw \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e DISPLAY=:0 \
-p 14556:14556/udp \
--name=container_name px4io/px4-dev bash
```

If everything went well you should be in a new bash shell now. Verify if everything works by running SITL for example:

```
cd <container_src>
make posix_sitl_default gazebo
```

Graphics driver issues

It's possible that running Gazebo will result in a similar error message like the following:

```
libGL error: failed to load driver: swrast
```

In that case the native graphics driver for your host system must be installed. Download the right driver and install it inside the container. For Nvidia drivers the following command should be used (otherwise the installer will see the loaded modules from the host and refuse to proceed):

```
./NVIDIA-DRIVER.run -a -N --ui=none --no-kernel-module
```

More information on this can be found here: <http://gernotklingler.com/blog/howto-get-hardware-accelerated-opengl-support-docker/>

Re-enter the container

If you exit the container, your changes are left in this container. The above "docker run" command can only be used to create a new container. To get back into this container simply do:

```
# start the container
sudo docker start container_name
# open a new bash shell in this container
sudo docker exec -it container_name bash
```

If you need multiple shells connected to the container, just open a new shell and execute that last command again.

Virtual machine support

Any recent Linux distribution should work.

The following configuration is tested:

- OS X with VMWare Fusion and Ubuntu 14.04 (Docker container with GUI support on Parallels make the X-Server crash).

Memory

Use at least 4GB memory for the virtual machine.

Compilation problems

If compilation fails with errors like this:

```
The bug is not reproducible, so it is likely a hardware or OS problem.  
c++: internal compiler error: Killed (program cc1plus)
```

Try disabling parallel builds.

Allow Docker Control from the VM Host

Edit `/etc/default/docker` and add this line:

```
DOCKER_OPTS="${DOCKER_OPTS} -H unix:///var/run/docker.sock -H 0.0.0.0:2375"
```

You can then control docker from your host OS:

```
export DOCKER_HOST=tcp://<ip of your VM>:2375  
# run some docker command to see if it works, e.g. ps  
docker ps
```

Legacy

The ROS multiplatform containers are not maintained anymore:

<https://github.com/PX4/containers/tree/master/docker/ros-indigo>

PX4 Continuous Integration

PX4 builds and testing are spread out over multiple continuous integration services.

Travis-ci

Travis-ci is responsible for the official stable/beta/development binaries that are flashable through [QGroundControl](#). It currently uses GCC 4.9.3 included in the docker image [px4io/px4-dev-base](#) and compiles px4fmu-{v1, v2, v4}, mindpx-v2, tap-v1 with makefile target qgc_firmware.

Travis-ci also has a MacOS posix sitl build which includes testing.

Semaphore

Semaphore is primarily used to compile changes for the Qualcomm Snapdragon platform, but also serves as a backup to Travis-ci using the the same [px4io/px4-dev-base](#) docker image. In addition to the set of firmware compiled by Travis-ci, Semaphore also builds for the stm32discovery, crazyflie, runs unit testing, and verifies code style.

CircleCI

CircleCI tests the proposed next version of GCC to be used for stable firmware releases using the docker image [px4io/px4-dev-nuttx-gcc_next](#). It uses the makefile target quick_check which compiles px4fmu-v4_default, posix_sitl_default, runs testing, and verifies code style.

Jenkins CI

Jenkins continuous integration server on [SITL01](#) is used to automatically run integration tests against PX4 SITL.

Overview

- Involved components: Jenkins, Docker, PX4 POSIX SITL
- Tests run inside [Docker Containers](#)
- Jenkins executes 2 jobs: one to check each PR against master, and the other to check every push on master

Test Execution

Jenkins uses [run_container.bash](#) to start the container which in turn executes [run_tests.bash](#) to compile and run the tests.

If Docker is installed the same method can be used locally:

```
cd <directory_where_firmware_is_cloned>
sudo WORKSPACE=$(pwd) ./Firmware/integrationtests/run_container.bash
```

Server Setup

Installation

See setup [script/log](#) for details on how Jenkins got installed and maintained.

Configuration

- Jenkins security enabled
- Installed plugins
 - github
 - github pull request builder
 - embeddable build status plugin
 - s3 plugin

- notification plugin
- collapsing console sections
- postbuildscript